Block Size Optimization for VBASBS Queueing Model for Blockchain

Nathan Crosby
dept. of Computer Science
Oklahoma State University
Stillwater, OK, USA
ncrosby@okstate.edu

Andrew Smith
dept. of Computer Science
Oklahoma State University
Stillwater, OK, USA
scu@okstate.edu

Christopher Crick
dept. of Computer Science
Oklahoma State University
Stillwater, OK, USA
chriscrick@cs.okstate.edu

Nohpill Park
dept. of Computer Science
Oklahoma State University
Stillwater, OK, USA
npark@cs.okstate.edu

***Abstract*—This paper demonstrates an algorithm utilized to increase batch processing efficiency of small blockchain transactions that are received online in a predictable manner. The specific implementation is a vague resemblance to crypto currency transaction processing but is meant to simulate a broader set of problems with all blockchain technology transactions. The idea is that creating a block to batch process transactions is expensive and so should not occur each time there is a new small transaction in the queue. It is unknown the size of the incoming blocks, or the time the next block will arrive, but there is an assumed relationship between the size of the block and the time lag in arrival time. We would like to control the size of the larger block that will batch process transactions. In the synchronous base case, we will assume that the size of this larger block is determined beforehand and is static. We propose an algorithm within the Variable Bulk Arrival and Static Bulk Service Queueing Model that adapts the size of the block between batch processing times. The optimization process seeks to decrease waiting time in the queue as much as possible without hurting throughput significantly.**

***Index Terms*—Blockchain, Optimization, Queueing**

I. Introduction

Over the last decade, crypto-currency has elevated the blockchain paradigm to the forefront of decentralized economic transfer schemes starting with the first widely accepted application, Bitcoin [6]. The proof-of-work concept has been adapted to other crypto-currency applications, most notably Ethereum [10]. Both of these mainstream crypto-currencies as well as most other alternative coins focus on the accuracy and redundancy of the transactions, with minimal effort given to reduce individual transaction waiting time and increase efficiency of processing transactions.

To this end, there is room for a sizable decrease in individual transaction waiting time if the size of the block processing the transactions can be altered dynamically. In this paper, we propose an algorithm to decrease waiting time while maintaining a high level of throughput. To begin the optimization process, we must first choose a performance model that analytically describes the relationship between the number of transactions, waiting time for those transactions to be processed, and the total number of transactions that are processed (throughput). Once a representative performance model has been chosen, an optimal level of block size can be selected that balances out the positive and negative aspects of the model, namely waiting time versus throughput.

Our paper is organized as follows. In section two we conduct a brief literature review and then focus on the performance model that will underly our own work. Section three is dedicated to the methodology we used to optimize the block size used in our simulations. In section four, we discuss the results recorded from the simulations run with a static block size versus our dynamic, optimized block size. In section five, we give concluding remarks and discuss future areas to continue the work on this path. Appendices follow the bibliography containing python code that can be used to replicate the graphs seen in the paper.

II. Related Work

Performance of blockchain transaction speed has been a concern since the inception of crypto-currencies. There have been several attempts to rectify this shortcoming in previous work. We see Fu [2] state that when considering blockchain optimization, the throughput or number of transactions handled by a blockchain system are important indicators of its performance. Mechkaroska [5] lays out some of the various methods that have been explored so far. They include increasing the block size to increase the number of transactions being processed, which is increasing throughput, and a technique called ”sharding” that is present in some alt-coins, namely Zilliqa [8] which is mentioned in the paper. While sharding is not currently possible in the performance model that we are considering, changing the size of the block is the path that seems most reasonable for the current, mainstream blockchain schemes.

In fact, a large number of articles recognize the fact that increasing the block size in blockchain transaction processing will necessarily increase throughput [3], [5], [9], but the idea is to blindly increase the size to a larger static value, not to dynamically alter the block size based on predicted flow of transactions. The hard fork in Bitcoin to create Bitcoin Cash was mainly predicated on this difference, with the increase in block size from 1MB on Bitcoin to 8MB on Bitcoin Cash.

Liu [4] set out to optimize a blockchain application for an Industrial Internet of Things (IIoT), and created a performance model to gauge waiting time and throughput. Then four

variables (block producers, consensus algorithm, block size, and block interval) were optimized to increase throughput. In the performance model proposed by Seol [7] block interval is allowed to fluctuate naturally as would be found in real life situations. Block producers and consensus algorithm are independent in the Seol model, so the similarity between Liu and the solution we present here is that we also use a dynamic block size to optimize our output, but the difference is that we are using the expected upcoming block interval λ and the fixed cost to create a block μ to scale the block size for the next set of transactions.

Fu [1] proposed optimizing the blockchain by utilizing a new schema with a hashing algorithm based on Proactive Reconfigurable Computing Architecture (PRCA). While we recognize that new architectures are being proposed to correct the perceived shortcomings of blockchain, we will limit ourselves to optimizing the performance within the given structure of the mainstream blockchain schema. In this way we can help to facilitate adoption of block size optimization techniques while the workings of alternate blockchain architectures are gaining consensus.

A. Performance Model

Our algorithm will assume the performance model proposed by Seol, Kancharla, Ke, Kim and Park - A Variable Bulk Arrival and Static Bulk Service Queuing Model for Blockchain [7]. The arrival rate of the individual blockchain transactions is variable, but not unpredictable. The arrival rate is assumed to vary with the size of the incoming transaction and the rate is denoted by the symbol λ.

The variable μ is defined to be the processing time of a block to be posted and purged as noted in Seol [7], and is assumed to be fairly static as stated in the model. Since this is assumed to be true, we will focus mainly on the performance measures of L_Q, W_Q and γ as described below.

The variable n is defined to be the total number of slots in the block. This will be used to define when a block (or batch of transactions) will be processed to minimize wait time while maximizing throughput.

P_i denotes the states with P_n denoting the maximum state before the block is processed. Given from the paper, the performance measure L_Q, which is the average number of customers in the queue, can be written as:

$$L_Q = \sum_{i=0}^{n} iP_i \tag{1}$$

Where the equation of the sum of i times state P at i is given as:

$$\sum_{i=0}^{n} iP_i = \sum_{i=0}^{n} i(q_i P_0(\sum_{j=1}^{i} j(\sum_{k=1}^{i-1}(\prod_{l=1}^{k-1} q_l)k) + i)) \tag{2}$$

But as can be noted from the code in the appendix of the paper which is responsible for plotting the associated graphs, this is equivalent to:

$$\sum_{i=0}^{n} i(q_i P_0(\sum_{j=1}^{i} j(\sum_{k=1}^{i-1}(\prod_{l=1}^{k-1} q_l)k) + i)) = nP_n \tag{3}$$

Where the state P at time n is written as:

$$P_n = \frac{\lambda}{\mu}\frac{n(n+1)}{2}P_0 \tag{4}$$

Therefore, L_Q can be written as:

$$L_Q = n * \frac{\lambda}{\mu}\frac{n(n+1)}{2}P_0 \tag{5}$$

The performance measure W_Q, which is the average amount of time a customer is in the queue, can be written as:

$$W_Q = \frac{L_Q}{\lambda} \tag{6}$$

Both of these performance measures will vary somewhat constantly by n times the ratio of $\frac{\lambda}{\mu}$ past some constant. And it can be noted that W_Q is simply a scaled measure of L_Q. However, the opposite end of this dynamic is the throughput, or how many transactions will be processed in a given amount of time. This is given in the model paper as γ, which is the throughput of the model, and is written as:

$$\gamma = \mu P_n = \mu\frac{\lambda}{\mu}\frac{n(n+1)}{2}P_0 = \lambda\frac{n(n+1)}{2}P_0 \tag{7}$$

These performance metrics based off of the model put forth in Seol [7] will form the basis of our algorithm. We will then try to minimize the waiting time while maximizing throughput based on the expectation of λ to adjust n to an optimum level. In doing so, we will compare our non-synchronous model to a synchronous model with a set value of block size n.

B. Python Code

To better understand the model presented in the paper, the MATLAB code available in the Appendix was reviewed and rewritten in Python. Only syntactical changes were made when converting the code base to Python.

This allowed us to examine some of the graphs in more detail. We can look at the graphs of γ, Fig.1, and W_Q, Fig.2, just to make sure we are on the same page as the paper with our given Python code.

As can be seen from the graphs in Fig. 1 and Fig. 2 the Python code replicates the graphs from Seol [7] exactly, therefore the performance model in our future optimization simulations is accurate according to their abstraction.

III. Methodology

A. Pseudocode

The proposed algorithm will incorporate the performance model Python code defined in Sec. II-B. The goal is to predict the optimal n based on all of the λs that have been seen by the model up until that point and then calculate the new n that will be used on the subsequent iteration.

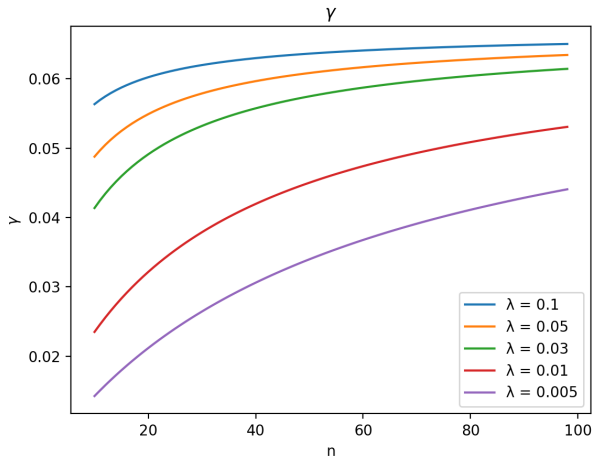

Fig. 1. Plot outputted from Model showing γ.

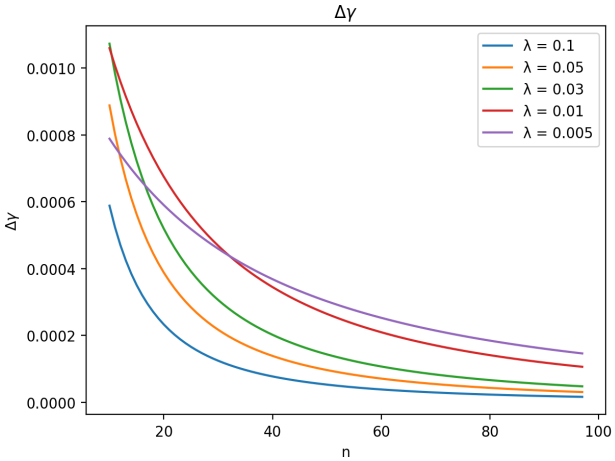

Fig. 3. Plotting output for $\Delta\gamma$.

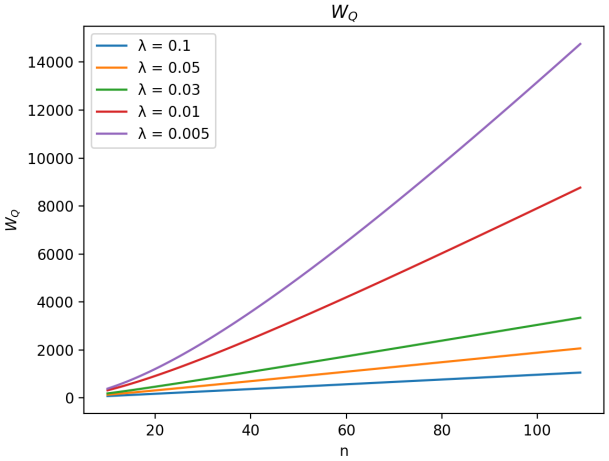

Fig. 2. Plot outputted from Model showing W_Q.

Algorithm 1 Process Blocks

Input: n, iter, lower, upper

nSyn ← n // make the sync block size n
nNsyn ← n // also start the non-sync block size as n
// set up arrays to hold the outputs
WQs ← zeros(iter)
LQs ← zeros(iter)
GammaS ← zeros(iter)
WQn ← zeros(iter)
LQn ← zeros(iter)
GammaN ← zeros(iter)
lambdaArr ← zeros(iter)
mu ← 1/15 // static as in model
for $i \leftarrow 1$ to iter: **do**
    lambda ← uniformRandom(lower, upper)
    (WQs[i], LQs[i], GammaS[i]) ← model(nSyn, 1, lambda, mu)
    (WQn[i], LQn[i], GammaN[i]) ← model(nNsyn, 1, lambda, mu)
    lambdaArr[i] ← lambda
    // optimize nNsyn based off the expected lambda
    nNsyn ← calcN(lambdaArr, mu)
end for

B. Optimization

The main point left to discuss is the optimization problem to set n to an optimum level given what the new expectation is for the coming λ. The new expectation for the coming λ will be the average of all λ that we have seen up until that point, which is an unbiased estimator of the random variable λ since our model will not have any prior knowledge of the range of values to be passed in.

The derivatives of the γ and W_Q are difficult to calculate for these purposes, so to approximate the derivatives we will use the first differences. This calculation is given by:

$$\Delta\gamma = \gamma[2:n] - \gamma[1:n-1] \tag{8}$$

$$\Delta W_Q = W_Q[2:n] - W_Q[1:n-1] \tag{9}$$

Since the functions are evaluated at small, discrete, and uniformly spaced time steps the approximation should be accurate for our purposes. Let's look at the visualization of the first differences for both γ as seen in Fig.3 and in W_Q as seen in Fig.4.

Looking at these first differences can give us some base understanding, but looking at the first difference for L_Q can give us some further insight as to what is happening with W_Q, pictured in Fig.5.

As can be seen, the first differences of L_Q are approaching one, meaning that the graph of L_Q is becoming linear after a certain level of n for a given value of λ demonstrated in the graph of L_Q shown in Fig.6.

Now, the difference between L_Q and W_Q is the factor $\frac{1}{\lambda}$. Therefore, as the first differences of L_Q are approaching one, the first differences of W_Q are approaching $\frac{1}{\lambda}$. Knowing this, and with the first differences of γ approaching zero, we can get an idea of how we need to optimize the value of n.

We need to traverse γ by increasing n until the first difference becomes low enough that the added benefit of

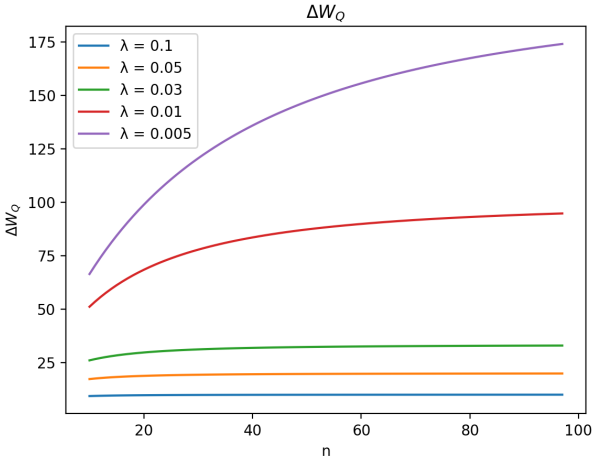

Fig. 4. Plotting output for ΔW_Q.

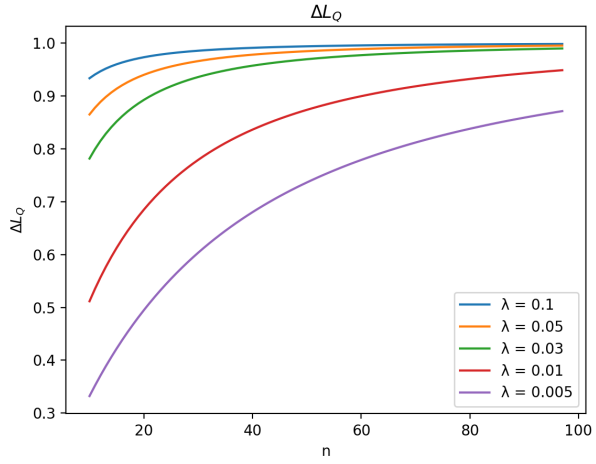

Fig. 5. Plotting output for ΔL_Q.

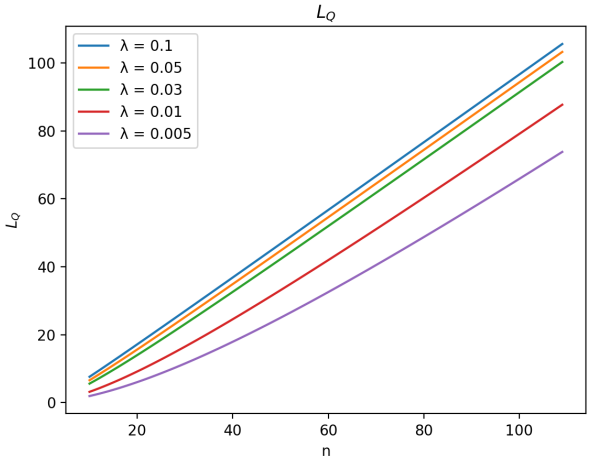

Fig. 6. Plotting output for L_Q.

extra throughput is outweighed by the negative effect of the increased waiting time. We will know that once the threshold is reached, the increase in n will only be increasing throughput by a small increment, but waiting time will be increasing linearly at that point. To capture this dynamic, we devised a metric that will take the ratio of change in γ over the change in W_Q times a penalty factor, which in our case is n*lg(n). This process can be described by the pseudo code documented below in Alg. 2.

Algorithm 2 calcN

Input: lambdaArr, mu
Output: optN

```
expLambda ← sum(lambdaArr) / sizeof(lambdaArr)
metric ← maxInt
n ← 10
optN ← n
(WQ, LQ, Gamma) ← model(n, expLambda, mu)
deltaWQ ← WQ[1:] - WQ[:-1]
deltaGamma ← Gamma[1:] - Gamma[:-1]
for i ← 0 to 89: do
    scale ← -(n + i) * lg(n + i)
    calcMetric ← scale * deltaGamma[i] / deltaWQ[i]
    if calcMetric < metric then
        metric ← calcMetric
        optN ← n + i
    else
        break
    end if
end for
return optN
```

It can be visualized in the graph seen in Fig.7 that the algorithm will traverse the graph for a given level of λ and stop when a minimum is reached. This algorithm takes advantage of the fact that the metric we have chosen decreases as γ is rising then increases as γ flattens out and W_Q continues to increase.

The equation for the optimization is given in Eq. 10 below for a given λ and μ with n in the range of 10 to 98 to avoid overflow errors in the current implementation of the performance model, with $\Delta\gamma$ and ΔW_Q defined in Eqs.8 and 9 above, respectively.

$$optN = argmin f(n) := \{-nlg(n) \cdot \frac{\Delta\gamma}{\Delta W_Q}\} \quad (10)$$

IV. Results

In implementing this algorithm, the results were as expected when the synchronous block size was set too high. In that scenario the incremental throughput (γ) was not justified for the given level of waiting time for transactions in the queue. Therefore, large decreases could be seen in both L_Q and W_Q for small relative decreases in throughput when adjusting the block size to a more appropriate level. With the penalty factor added in (n lg(n)), whose purpose is to push the optimal block

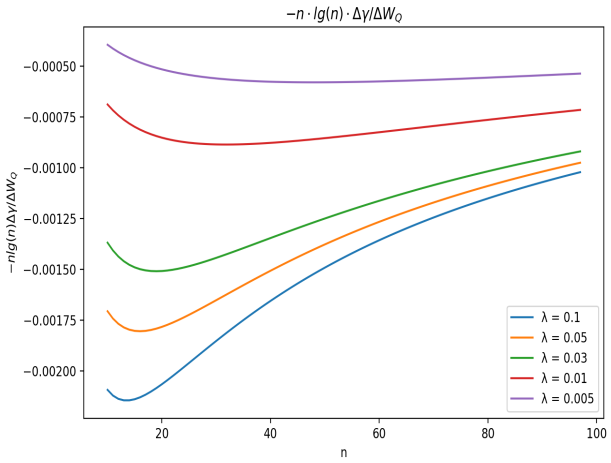

Fig. 7. Plotting output for $-n \cdot lg(n) \cdot \Delta\gamma/\Delta W_Q$.

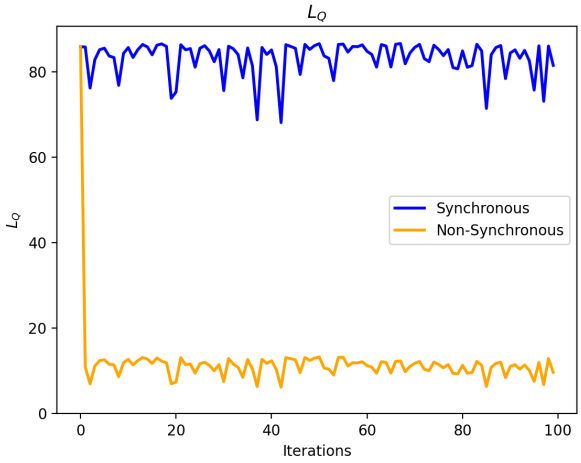

Fig. 8. Plotting output showing L_Q performance.

size back out on the curve, this gives the opposite effect when the synchronous block size is set too low. Without the penalty factor the optimization algorithm will peg the optimal block size to the lowest bound allowed in the algorithm. Since this result is not the intention, the penalty factor needs to be left in, but since the power of the algorithm is only on the side of reducing the block size a cap should be implemented that restricts the optimal block size to be less than or equal to the synchronous block size for any expected λ.

In tables I and II we present the output from our optimization method with the number of random draws of λ being one hundred. We divided the results in the tables into three categories, namely a high, medium and low static block size. Within each category there are runs for a wide range of λ, a narrow/high range of λ, and a narrow/low range of λ to try to capture the scenarios that would be seen when running this process.

As an example of how this process plays out, the graphs below are from a run with the synchronous block size set to 90 with the λ using the largest range of 0.005 to 0.1 and a uniform random draw with mu set to 1/15.

V. Conclusion and Discussion

In this paper we have demonstrated that if the queueing model for blockchain transaction processing follows the VBASBS model described in Seol [7], that the block size optimization process described can drastically reduce transaction waiting time in the queue while preserving a majority of the throughput if the synchronous block size is deemed to be too large.

Future work can focus on the expected λ prediction method. In this paper there was no assumption made on how the transaction arrival rate progressed through time. Therefore, the best estimate for the future value of λ is equal to the average of all observations of λ in the past. However, if there is strong evidence that transaction arrival rate evolves in a

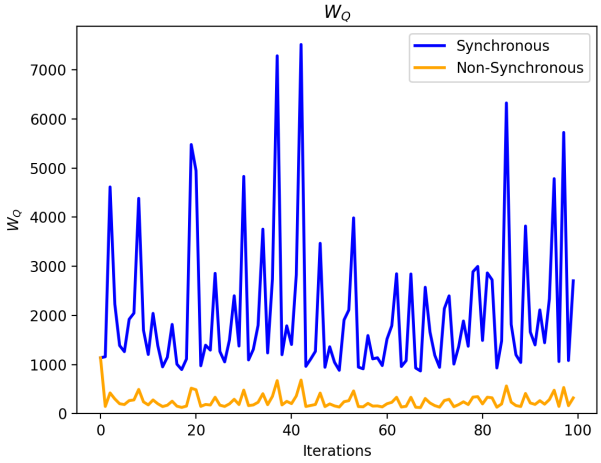

Fig. 9. Plotting output showing W_Q performance.

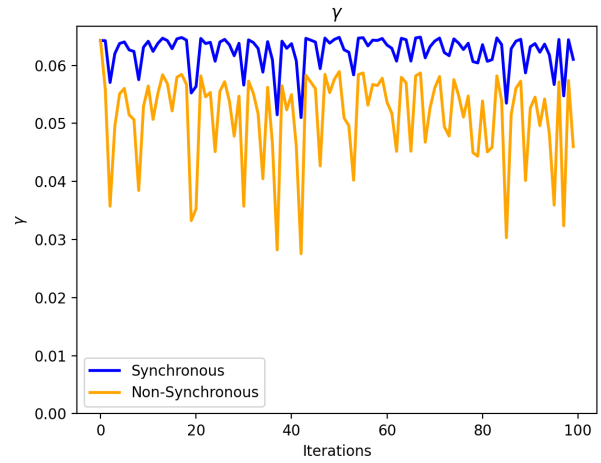

Fig. 10. Plotting output showing γ.

TABLE I
TABLE SHOWING PERFORMANCE GAINS WITH $\mu = 1/15$.

All output averaged over 100 blocks

		mu = 1/15 uniform draw								
		Static N			Non-Static N			Percent diff		
	lamda range	LQ	WQ	Gamma	LQ	WQ	Gamma	LQ	WQ	Gamma
Static N = 90	0.005-0.1	85.578	2335.844	0.064	7.776	285.888	0.052	-91%	-88%	-19%
	0.05-0.1	87.290	1167.093	0.065	8.545	114.644	0.057	-90%	-90%	-12%
	0.005-0.01	74.669	9781.027	0.056	11.048	1414.024	0.037	-85%	-86%	-34%
Static N = 50	0.005-0.1	45.994	1149.776	0.063	7.428	182.162	0.052	-84%	-84%	-17%
	0.05-0.1	47.310	646.591	0.064	8.121	110.399	0.057	-83%	-83%	-11%
	0.005-0.01	35.782	5055.734	0.049	11.074	1544.637	0.036	-69%	-69%	-26%
Static N = 15	0.005-0.1	11.500	300.016	0.055	7.007	176.645	0.051	-39%	-41%	-6%
	0.05-0.1	12.588	169.468	0.060	7.788	104.592	0.057	-38%	-38%	-4%
	0.005-0.01	6.539	895.329	0.031	6.539	895.329	0.031	0%	0%	0%

TABLE II
TABLE SHOWING PERFORMANCE GAINS WITH $\mu = 1/5$.

All output averaged over 100 blocks

		mu = 1/5 uniform draw								
		Static N			Non-Static N			Percent diff		
	lamda range	LQ	WQ	Gamma	LQ	WQ	Gamma	LQ	WQ	Gamma
Static N = 90	0.005-0.1	79.399	2184.255	0.178	8.103	173.809	0.122	-90%	-92%	-32%
	0.05-0.1	83.869	1167.193	0.188	7.663	106.792	0.138	-91%	-91%	-27%
	0.005-0.01	55.410	7641.615	0.125	18.386	2502.940	0.088	-67%	-67%	-29%
Static N = 50	0.005-0.1	40.604	1070.489	0.166	7.295	165.533	0.119	-82%	-85%	-28%
	0.05-0.1	44.268	603.365	0.181	7.272	99.006	0.138	-84%	-84%	-23%
	0.005-0.01	23.590	3239.721	0.096	18.802	2578.471	0.089	-20%	-20%	-7%
Static N = 15	0.005-0.1	8.995	200.191	0.129	7.661	169.565	0.124	-15%	-15%	-4%
	0.05-0.1	10.532	138.706	0.150	6.990	91.799	0.139	-34%	-34%	-7%
	0.005-0.01	3.149	434.035	0.045	3.149	434.035	0.045	0%	0%	0%

specific pattern, such as a cyclical pattern with strong mean reversion, then that process could be incorporated into the expected λ calculation.

Based on these results, the efficiency of the VBASBS queueing model can be improved dramatically with active management of the size of the blocks to be processed, and the method presented here facilitates that process with very meaningful results.

REFERENCES

[1] J. Fu, S. Qiao, Y. Huang, X. Si, B. Li, and C. Yuan, "A study on the optimization of blockchain hashing algorithm based on prca," *Security and Communication Networks*, 2020.

[2] X. Fu, R. Yu, J. Wang, Q. Qi, and J. Liao, "Performance optimization for blockchain-enabled distributed network function virtualization management and orchestration," *IEEE Transactions on Vehicular Technology*, 2020.

[3] J. Göbel and A. E. Krzesinski, "Increased block size and bitcoin blockchain dynamics," in *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*. IEEE, 2017, pp. 1–6.

[4] M. Liu, F. R. Yu, Y. Teng, V. C. Leung, and M. Song, "Performance optimization for blockchain-enabled industrial internet of things (iiot) systems: A deep reinforcement learning approach," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 6, pp. 3559–3570, 2019.

[5] D. Mechkaroska, V. Dimitrova, and A. Popovska-Mitrovikj, "Analysis of the possibilities for improvement of blockchain technology," in *2018 26th Telecommunications Forum (TELFOR)*. IEEE, 2018, pp. 1–4.

[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Bitcoin White Paper*, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[7] J. Seol, A. Kancharla, Z. Ke, H. Kim, and N. Park, "A variable bulk arrival and static bulk service queueing model for blockchain," in *Proceedings of the 2nd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, 2020, pp. 63–72.

[8] Z. Team *et al.*, "The zilliqa technical whitepaper," *Zilliqa Whitepaper*, 2017.

[9] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 264–276.

[10] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

APPENDIX A
PERFORMANCE MODEL

Python code to run the performance model in Seol [7]. Comments left out for brevity and to reduce redundancy from explanations above.

```
import numpy as np

def blockchain(n, iterations, lamda, mu):
  L = np.zeros(iterations)
  W = np.zeros(iterations)
  WQ = np.zeros(iterations)
  LQ = np.zeros(iterations)
  Gamma = np.zeros(iterations)

  for iter in range(0, iterations):
    Q = np.zeros(n, dtype=np.longdouble)
    a = np.zeros(n, dtype=np.longdouble)
    P = np.zeros(n, dtype=np.longdouble)
    temp = np.ones(n, dtype=np.longdouble)

    for i in range(1, n):
      temp[i - 1] = 2 ** (i - 1) * \
        (i * n - 2 * n + 7 * i - 14 - \
        i ** 2) + 8
      a[i - 1] = i * ((i + 1) / 2) * \
        (n / (np.sqrt(2 * np.pi * n) * \
        (n / np.e) ** n) ** 2) * \
        temp[i - 1] + i
      Q[i - 1] = a[i - 1] * \
        (2 / ((n - i) * (n - i + 1)))

    Psum = np.sum(Q) + 1 + (lamda / mu) \
      * ((n * (n + 1)) / 2)
    Pzero = 1 / Psum

    P[n - 1] = (lamda / mu) * \
      ((n * (n + 1)) / 2) * Pzero

    LQ[iter] = (n - 1) * P[n - 1]

    WQ[iter] = LQ[iter] / lamda
    W[iter] = WQ[iter] + 1 / mu
    L[iter] = W[iter] * lamda
    Gamma[iter] = P[n - 1] * mu

    n += 1

  return (LQ, WQ, Gamma)
```

APPENDIX B
SIMULATION

Python code to run Alg.1 from Sec.III-A. Comments left out for brevity and to reduce redundancy from explanations above.

```
import numpy as np
import Algorithm as alg
import Queueing_Model as qm

def process(draw, graphOut, avgOut):
  numPasses = 100
  mu = 1.0 / 15.0
  low = 0.005
  high = 0.1
  mean = 0.005
  stdev = 0.001
  nSyn = nNsyn = 90
  lamdaList = list()

  IterLQSyn = np.zeros(numPasses)
  IterWQSyn = np.zeros(numPasses)
  IterGammaSyn = np.zeros(numPasses)
  IterLQNsyn = np.zeros(numPasses)
  IterWQNsyn = np.zeros(numPasses)
  IterGammaNsyn = np.zeros(numPasses)
  IterNNsyn = np.zeros(numPasses)

  for i in range(0, numPasses):
    lamda = 0.0
    if draw == 'uniform':
      lamda = np.random.uniform(low, high)
    elif draw == 'gaussian':
      lamda = np.random.normal(mean, \
        stdev)

    (LQSyn, WQSyn, GammaSyn) = \
      qm.blockchain(nSyn, 1, lamda, mu)
    (LQNsyn, WQNsyn, GammaNsyn) = \
      qm.blockchain(nNsyn, 1, lamda, mu)

    IterLQSyn[i] = LQSyn[0]
    IterWQSyn[i] = WQSyn[0]
    IterGammaSyn[i] = GammaSyn[0]

    IterLQNsyn[i] = LQNsyn[0]
    IterWQNsyn[i] = WQNsyn[0]
    IterGammaNsyn[i] = GammaNsyn[0]

    IterNNsyn[i] = nNsyn

    lamdaList.append(lamda)
    nNsyn = alg.calcN(lamdaList, mu)
    if nNsyn > nSyn:
      nNsyn = nSyn

  if graphOut:
    alg.plotting('$L_Q$', \
      np.arange(numPasses), IterLQSyn, \
      IterLQNsyn)
    alg.plotting('$W_Q$', \
```

```
      np.arange(numPasses), IterWQSyn, \
      IterWQNsyn)
    alg.plotting('$\gamma$', \
      np.arange(numPasses), \
      IterGammaSyn, IterGammaNsyn)

  if avgOut:
    print('(' + draw + ')average LQ: %f,\
      %f'%(np.average(IterLQSyn), \
      np.average(IterLQNsyn)))
    print('(' + draw + ')average WQ: %f,\
      %f'%(np.average(IterWQSyn), \
      np.average(IterWQNsyn)))
    print('(' + draw + ')average Gamma: \
      %f, %f'%(np.average(IterGammaSyn),\
      np.average(IterGammaNsyn)))
    print('(' + draw + ')average N: %f, \
      %f'%(nSyn, np.average(nNsyn)))

if __name__ == '__main__':
  process('uniform', False, True)
  process('gaussian', False, True)
```

APPENDIX C
OPTIMIZATION

Python code to run Alg.2 from Sec.III-B. Comments left out for brevity and to reduce redundancy from explanations above.

```
import numpy as np
import matplotlib.pyplot as plt
import Queueing_Model as qm

def calcN(lamdaList, mu):
  expectedLamda = sum(lamdaList) / \
    len(lamdaList)

  metric = float("inf")
  n = 10
  optN = n
  numIter = 89

  (LQ, WQ, Gamma) = qm.blockchain(n, \
    numIter, expectedLamda, mu)

  deltaWQ = WQ[1:] - WQ[:numIter-1]
  deltaGamma = Gamma[1:] - \
    Gamma[:numIter-1]

  for i in range(0, numIter-1):
    calcMetric = -(n + i) * \
      np.log2(n + i) * deltaGamma[i] / \
        deltaWQ[i]

    if calcMetric < metric:
      metric = calcMetric
      optN = n + i
    else:
      break

  return optN

def plotting(title, x, ySyn, yNsyn):
  plt.plot(x, ySyn, color='b', \
    label="Synchronous", linewidth=2)
  plt.plot(x, yNsyn, color='orange', \
    label="Non-Synchronous", linewidth=2)

  plt.title(title)
  plt.ylim(bottom=0)
  plt.xlabel("Iterations")
  plt.ylabel(title)
  plt.legend(loc='best')
  plt.show()
```