A Model of Completion Queue Mechanisms using the Virtual Interface API

Naresh Patel*
Network Appliance, Inc.
naresh@netapp.com

Hari Sivaraman
Compaq Computer Corp.
hsivaram@acm.org

Abstract

In this paper we evaluate the performance of different completion mechanisms for the Virtual Interface (VI) Architecture from an application perspective. Based on this comparison we show that the completion queue model offers an optimal[1] way to receive messages in multi-threaded applications under moderate to heavy traffic loads. The application model presented in this paper demonstrates efficient ways of using the VI interface.

1 Introduction and Background

The Virtual Interface (VI) Architecture [1][2][8] provides a cluster communication mechanism that allows user processes to communicate without executing any kernel-mode code. VI provides protected user level communication with low latency and low CPU cost per message as compared to communication protocols like TCP/IP and UDP. The VI architecture improves latency and CPU cost per message by reducing the need to make user/kernel transitions and by making it possible for the Network Interface Card (NIC) to access user memory directly.

VI is made of three components and an application. The three components are the VI interface, the VI provider and the Completion Queue (CQ) The VI interface is the mechanism used by the application to perform data transfer operations. A VI interface consists of a pair of queues, a send queue and a receive queue, and a pair of doorbells. Each queue owns a doorbell. When an application needs to receive or send data it posts a descriptor on the corresponding queue and rings a doorbell to notify the VI adapter that it has posted a descriptor. The VI provider processes the descriptor asynchronously and then sets a status bit to notify completion.

The VI provider is a combination of hardware and software that provides the communication model that is abstracted by a VI interface. The VI provider consists of a Network Interface Controller (NIC), and a Kernel Agent (KA). The NIC provides the send and receive queues, the CQs and the actual data transfer. The KA is typically a driver that performs connection setup, tear down, interrupt handling, error handling, and memory management. The KA is never involved in error-free data transfers.

The application represents an end-point of the VI architecture. An application may access the VI API directly by linking to a library. The library makes calls to the KA to create VIs, tear down VIs, manage memory and set up CQs. The library also abstracts the procedure for enqueuing and de-queuing descriptors from the queues in the VI interface. The library is shown in Figure 1 as the VI User Agent.

Once a connection is made between, two nodes in a cluster data can be exchanged using one of two different modes: *send-receive* mode or *RDMA (Remote DMA)* mode. The send-receive mode of data transfer provides typical channel-based semantics. In this mode, the receiver posts a descriptor to receive the data that it is going to receive. A descriptor is a data structure that has among other things a pointer to a buffer for data. The size of the receive buffer must be the same size or bigger than the data that is to be received. The sending side specifies the location of the data that is to be sent. The send side then does a post of the data that is to be sent. This mode requires that both the sender and the receiver be notified of completion at both ends of the transfer.

---

*Work done while at Compaq Computer Corporation

[1]Optimal in the sense of minimizing the overall CPU overhead per message transfer. Latency and throughput are other performance metrics of interest.

0-7695-0896-0/00 $10.00 © 2000 IEEE

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

Google Exhibit 1049
Google v. VirtaMove
Oracle Ex. 1049
Oracle v. VirtaMove, IPR2025-00982

VI also provides a mechanism to transfer data using memory semantics using RDMA. In this mode, the sender (better called the initiator) specifies both the source and destination buffer for the data transfer. There are two types of RDMA operations: RDMA Read and RDMA Write. In an RDMA Read operation the initiator specifies the source of the transfer on the remote node and the destination on the local node. In an RDMA Write operation the initiator specifies the source of the transfer in a local node and the destination on the remote node. On completion of RDMA operations no notification is given to the remote node. If the remote node needs notification then a *RDMA write with immediate data* can be used. If immediate data is used with an RDMA operation it would consume a pre-posted buffer on the remote node and both the initiator and the remote node will be notified of completion.

The VI architecture provides several mechanisms to notify completion of a descriptor. The mechanisms on a send or receive queue are *poll*, *wait*, and *notify*. In a poll mode the application continually checks (i.e polls) a *status bit* to determine whether the VI provider has completed work on a descriptor. In the Wait mode the application goes to sleep on the send or receive queue and is woken up when the VI provider has completed the descriptor. In the notify mode the application hands to the VI provider a call-back function that is executed once the VI provider has completed the descriptor.

A CQ can also be used to inform the application of a completed request. Once a VI is created, the send and/or receive queue may be associated with a completion queue. Once a send or receive queue is associated with a completion queue all completion notification must be handled on the completion queue. There are three mechanisms for notification on a completion queue; i.e *poll, wait* and *notify*. These mechanisms are identical to those on send and receive queues. An application may associate a large number of VIs with the same completion queue. This provides a mechanism to aggregate all the completion events for the VIs on one queue.

Therefore from an application standpoint there are six different mechanisms to indicate completion: poll, wait, notify, CQ-poll, CQ-wait, and CQ-notify. These six mechanisms fall into two classes: interrupt-based (wait and notify) and poll-mode based. Since the polling mechanism wastes CPU time by busy spinning until a completion occurs, it is generally unattractive for transaction processing systems[2]. In this paper we focus primarily on various interrupt-based mechanisms. We first present a queueing model to analyse the performance of an interrupt-based completion mechanism. The results of this model suggest that a CQ-wait method would be optimal. Based on this result we developed an application model to compare these different completion mechanisms and to demonstrate that completion queue waits deliver messages with the lowest CPU overhead and latency.

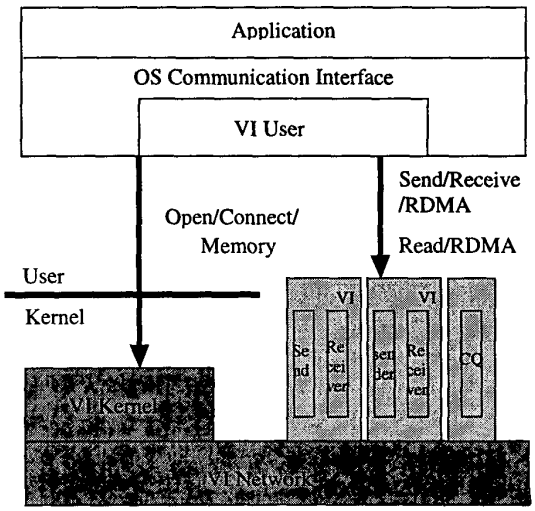

Figure 1: The VI Architectural Model [2]

The metric that we use for this comparison is the CPU cost. It is defined as the total number of microseconds spent by all CPUs (all processors at the sender and receiver) divided by the number of messages transferred during an interval. For example, suppose the CPU cost to send and receive a message of size 8 bytes is 20 microseconds (μs). This means that overall, it takes 20 μs of CPU time to execute the instructions to both send and receive a message. Frequently, this includes software to start the transfer, and initiate PCI transfers to send the message. Once the message is received, it includes the time to run an interrupt service routine, and execute the code to collect the message. It also includes any OS activity such as context switches that occur

[2] An exception to this might be in those cases where the application runs on an SMP and the application threads are statically scheduled so that one or more processors are dedicated to messaging activity. In such a scenario it would be optimal to poll on the processors that are dedicated to messaging.

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

during the send and receive process. Figure 2 shows a representation of this definition.

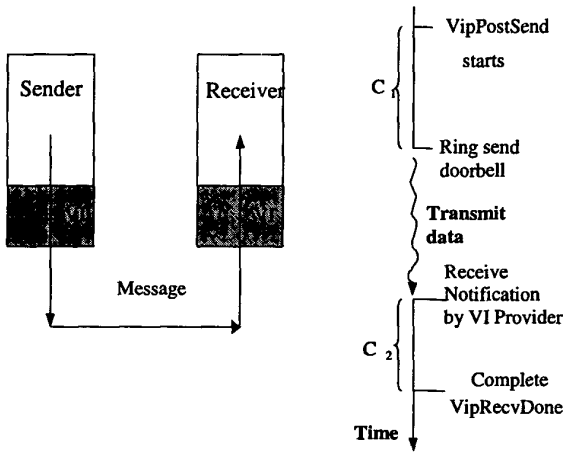

Figure 2: This shows a schematic of a message sent from a sender to a receiver. The figure on the right shows a time sequence of the events that go to make up the CPU cost to send-receive a message. The CPU cost is the sum of C_1 and C_2. The CPU cost is defined as though one CPU does both the send and the receive. It might also be the case that the components C_1 and C_2 might overlap in time but that does not alter the definition. Also, C_1 activity may continue after ringing the doorbell.

To compare the performance of different completion mechanisms from an application perspective we developed a model of an application over VI. In this paper we present our model and the results obtained from that model. Our models were developed in the context of VI in which users have an option to handle completions directly through a descriptor or use a CQ. However, these models for CQs are also valid for InfiniBand [12]. In the InfiniBand standard all completion notifications have to go through a CQ.

In the next section we discuss related work and present the motivation for this paper. In section 3 we present a queuing model and some closed-form formulas for the expected amount of interrupt aggregation and mean response time. In section 4 we present the application model that we used to generate the comparison data and in section 5 we present our results. The conclusion is presented in section 6.

2 Motivation and Related Work

User-mode communication mechanisms [1][3][4][5][8][13] show low latencies for data transfer. Frequently, this low latency is obtained using a polling mechanism. The polling mechanism expends CPU cycles waiting for a message as described by Speight et al [3]. The CPU cost (processor cycles) to send and receive a message is important because at a given message rate, it determines the amount of CPU time that remains for doing computation or useful database transactions. Therefore, even if the message itself is transferred very quickly with low latency, the application performance, *e.g.* number of database transactions executed per second could be low because CPU cycles are wasted on polling. An upper bound on the number of transactions, T_x could be obtained based on the CPU capacity of the system, computation and per-message cost. For example, in an n-processor system, if each transaction has a computation cost of C_P µs, a per-message cost of C_M µs and a message to transaction ratio of M an upper bound on the number of transactions, T_x would be

$$T_x \leq \frac{n10^6}{C_P + M\,C_M} \qquad (1)$$

Hence, from an application standpoint the CPU cost per message, C_M µs, is as important a metric as latency. VI provides an interrupt-based mechanism as an alternative to polling. This also suffers a CPU overhead in that each message runs an interrupt handler, a way to signal the application and possibly a couple of context switches. On a per message basis this can be a significant overhead, as shown in Table 3 in section 5, that could rob VI of much of its touted benefit. So, even though VI might offer latencies under 10 µs, the CPU overhead due to message completion can result in poor application performance. In this paper we analyse an interrupt-based completion mechanism and based on our analysis we present a model that demonstrates how an application may use VI with minimal CPU overhead per message and therefore achieve a high transaction rate or an increased amount of useful work. Speight et al [3] have presented a comparison of CPU utilization for polling and blocking. In constrast, our paper also discusses CQs, compares different blocking schemes, explores how an application should use a blocking mechanism to achieve high performance. We are not aware of any other published work that compares the CPU cost of completion mechanisms in VI from an application perspective.

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

3 Analytical Model of a Completion Mechanism

In this section we outline a model of a system in which messages use an interrupt-based mechanism to complete. First, we outline some basic assumptions that enable a tractable model.

- Messages arrive according to a Poisson[3] process with rate λ.
- Messages that arrive to an empty queue have a service time denoted by the random variable R with mean m_R and second moment M_{2R}.
- Messages that arrive to a non-empty queue have a service time denoted by the random variable B with mean m_B and second moment M_{2B}.
- A single Thread handles message completions and some application processing.

The random variable B represents the time to execute the completion thread for a single message when the thread has pending messages in the input queue. The random variable R is the sum of these three random variables:

1. R_{RUPT} which represents the time taken for an interrupt to signal a thread
2. R_{WAKE} which represents the time taken for a thread to begin execution after it is signaled
3. B which is defined above

In other words, R_{RUPT} and R_{WAKE} represent the overhead incurred when a message arrives to an empty queue.

Our goal here is to derive an expression for the average amount of aggregation that we can expect for a given load and service time distributions. First we need some definitions:

- Let p_n be the steady-state probability that n messages are in the input queue. This is the same as the probability that a departing customer leaves n customers behind it and also the same as the probability that a new arrival finds n customers in the queue (see [6] for more details).
- Let E[N] be the average number of messages aggregated in a single interrupt
- Let E[W] be the average response time for a message

p_0 is the probability that input queue is empty and since Poisson arrivals "see" time averages this is also the probability that an arrival will find the queue empty and cause an interrupt. Given an arrival rate λ, the interrupt rate due to messages is $\lambda\, p_0$ and so $E[N] = 1/p_0$.

The derivation of an expression for p_0 is given in Appendix A and this leads to a simple expression for E[N]:

$$E[N] = (1 + \lambda m_R - \rho)/(1 - \rho), \text{where} \quad \rho = \lambda m_B \qquad (1)$$

Note that the queue is stable if $\rho<1$ and stability is not dependent on m_R.

Suppose, $m_R = m_B + \gamma$ where γ is sum of the expected values of the random variables R_{RUPT} and R_{WAKE}. Hence equation (1) becomes

$$E[N] = (1 + \lambda\gamma)/(1 - \lambda m_B) \qquad (2)$$

From equation (2) we see if m_B is close to zero the interrupt aggregation is restricted to that which can be achieved by an interrupt handler; so this suggests that an application should try to achieve a reasonable value of m_B. Too high a value of m_B would result in high latencies but too low a value would result in low aggregation. A high value of λ results in a large aggregation hence the application benefits from high message rates. Since the number of messages to a VI cannot be changed by the application, a reasonable way to achieve a large value for λ is to aggregate the completions of different VIs on a single queue. Hence from equation (2) it can be seen that a CQ based mechanism would help achieve a high value for λ and a reasonable choice of value for m_B would result in high aggregation without increasing the latency excessively. Based on this result we built a model for an application that evaluates different completion mechanisms to find the one that is optimal in terms of CPU cost. The application model also allows the value of m_B to be varied so as choose a value that achieves high interrupt aggregation.

The components of the random variable R are affected by high loads in different ways. The mean of R_{RUPT} will be high under high interrupt-level processor activity, especially on Windows NT because of deferred procedure call (DPC) queueing time delays. Generally, the mean of R_{WAKE} will be even higher under high CPU load at the same or higher thread priority level than the

[3] In general, the message arrival process will not be Poisson, but we can represent more bursty traffic (approximately) by using a Markov-modulated Poisson process.

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

completion thread, since this time includes the queuing delay for obtaining a processor. The random variable B is also affected by high priority traffic under high loads that may pre-empt the CQ thread.

The mean response time for a message is given by the following expression (see Appendix B for a sketch of the derivation):

$$E[W] = \frac{p_0(2m_R - \lambda^2(M_{2B}m_R + M_{2R}m_B) + \lambda(M_{2R} - 2m_R m_B))}{2(\lambda m_B - 1)^2} \quad (3)$$

This is an extension of the Pollaczek-Khintchine result for M/G/1 queues. It shows that the queuing delay is affected by the first and second moments of the two service times only (and not any higher moments). If the two service time distributions are the same, this expression simplifies to the standard M/G/1 formula. In the next section, we describe an application model that was developed to validate the result that quantifiable performance gains are possible by using CQs.

4 Application Model that Represents Multi-threaded Applications

We developed a program to model the behavior of applications over VI that use a *request-response* model of communication. The motivation for using this model is that SAP R/3 ERP applications and numerous client to database or application server to database server applications use a *request-response* model. This application model allows the user to vary independently, the number of VI's, the number of threads, the number of CQs, and the mode of completion. We ran this application model to compare the performance of different modes of completion. Based on our results we recommend the completion mode that provides optimal performance under high message traffic conditions.

The model we used is a multi-threaded request-response test over native VI. The request-response test is a simple ping-pong test between VIs on the server and the client. The mechanism by which a message is received is defined by a user-specified parameter. Messages sent and received over each VI can be "*completed*" using one of three modes; *poll, wait* and *notify*. Further, the messages can either be completed by the VI directly or using a CQ with which the VI is associated. In addition when using a Wait mode on CQ it is possible for the application to either wait directly on the CQ (called CQWait mode) or create dedicated thread(s) that service the CQ (called CQWaitD mode). This dedicated thread is labeled CQThread. Therefore there are seven different ways in which a send or a receive operation may be completed.

In addition, a send operation may be completed using a *lazy mechanism*. In this mechanism, after posting a send descriptor we do not wait to collect the send descriptor. We postpone collecting the send descriptor until we have posted, say ten messages. Then we "complete" all those send descriptors that correspond to the messages that have been sent. This lazy mechanism helps reduce the CPU overhead to send messages. In our latency test we use the lazy mechanism to complete the send operations. Hence in our application model the user can choose any of the seven completion mechanisms to complete a receive operation.

Each CQThread can either poll or wait for completions on the CQ that it services. In the Wait mode each CQThread waits by sleeping on its CQ. Once it wakes up it picks up the VI from the CQ, goes to that VI, picks up the descriptor that completed, busy loops for a user-specified duration[4], and then wakes up the application thread to which the VI belongs. At this point it checks the CQ to see if any more completions have occurred. If there are any it drains the CQ and only then it goes to sleep on the queue. Pseudo-code for the core of the application model can be made available.

In the application model, in every completion mode the user can request that the thread that picks up the receive descriptor *busy-loop* for a specified duration. Typically, in a real application, the thread that picks up the message has to process the message before it can go on to send or receive another message[5]. The purpose of introducing this *busy-loop* is so that the user can accurately model the behavior of an application. All the results that are presented in this paper have a busy-loop duration of 40 μs. We experimented with different values for this "busy loop duration". We found that values greater than 40 μS do not achieve greater interrupt aggregation. A value for m_B that achieves reasonable interrupt aggregation[6] could be any duration over 25 μs.

Since the polling mechanism wastes CPU time by busy spinning on a CQ it is unattractive from an application perspective. Hence we have not presented any results using polling on a CQ. Alternately, one may use a notify mechanism on a CQ. In section 5 we explain why we do not consider the notify mechanism.

---

[4] This corresponds to the parameter m_B in section 3.

[5] This delay mimics time spent by a transport stack to process a message.

[6] An aggregation of 6:1 or greater

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

5 Results

We ran the application model on a pair of Compaq Proliant 6400r servers running Windows NT-2000 (build 2128). The servers were interconnected using a pair of ServerNet II NIC's. ServerNet II is Compaq's implementation of the VIA standard. Based on the CPU cost results from single-threaded tests we chose the modes of completion that were most promising and ran tests to compare their performance in a multi-threaded environment with moderate to heavy traffic loads. These tests showed that the Wait mode on a CQ with a separate thread to service the queue was optimal in terms of the CPU cost to send and receive a message.

We first compared the performance of different completion methods by running the application model for different message sizes with one application thread and one VI. The results, averaged out over five runs for each completion method, are presented in Table 1 below. From the data we can see that the one-way latency for a message completed using a Wait mode is about 20 µS greater than a Poll mode latency. The difference is the time it takes Windows 2000[7] to context switch the application thread into the CPU when it wakes up and to context switch it off the CPU when it sleeps on a receive. In addition, in the Wait mode, the VI interrupts need to be turned on when VipRecvWait is called and turned off once the interrupt has been dequeued from the interrupt queue. However, it turns out that sometimes, but not always, this operation is skipped depending on the relative timing of the two operations, and we call this "interrupts staying stuck on ON". The Wait mode on a CQ without an extra thread, called the CQWait mode, is almost identical to the Wait mode and this can be seen from the latency and CPU costs in Tables 1 and 2. Since the the wait and CQWait modes are almost identical we do not consider the CQWait mode anymore. We restrict ourselves to the Wait mode for purposes of comparison. The Wait mode on a CQ with an extra thread, called CQThread, to service the CQ adds an extra context switch into and out of the CQThread. This extra cost works out to 20 µs. Hence we expect the CPU cost to be 20 µs higher than in a Wait mode. But, we see from the data in Table 2 that it is higher. This is due to the cost of turning interrupts on and off. In the case of the Wait mode on a CQ with an extra thread, owing to the round-trip latencies, it is almost certain that the interrupts do not stay *stuck on ON* hence the latencies and the CPU cost reflect this added cost. The components of the CPU cost, broken up on a functional basis, are shown in

Table 1: This shows the one-way latency in µs to transfer messages over VI using different modes of completion. This was measured using a ping-pong test for messages ranging in size from 8 bytes to 65536 bytes and averaged over five trials The poll mode data falls along a straight line defined by the parameters "L" and "D"[8]. The symbol I in the rows labelled CQWaitD and Notify represents the time to turn interrupts on and off. This is described and defined in Table 3. L = 0.0128 D = 10.89

Mode	Latency (µS)	
Poll	L*(Message Size) + D	L =
Wait	L*(Message Size) + D + 20	0.0128
CQWait	L*(Message Size) + D + 20	D =
CQWaitD	L*(Message Size) + D + 20 + I	10.89
Notify	L*(Message Size) + D + 20 + I	

Table 2: This table shows the CPU cost in µS to send and receive a message over VI using different modes of completion. This data was measured using a ping-pong test.. The poll mode data falls along a straight line defined by the parameters "m" and "C"[9]. The symbol "W" denotes the CPU cost for a Wait mode and the symbol I is the same as in Table 3. The symbol N captures the overhead due to the Notify mechanism.

Mode	CPU cost	CPU cost(µS)
Poll	m*(Message Size) + C	m=0.024
		C = 25
Wait	$W = I_p + 20$	28
CQWait	W	28
CQWaitD	W + 20 + I	63
Notify	W + 20 + I + N	80

Table 3. From these components it is possible to make a fairly accurate estimate of the CPU cost for any completion method. Such an estimate is often only fairly accurate because of two reasons. First, interrupts sometimes get stuck on ON, and second, the cost of servicing an interrupt often gets amortized over a number of messages whenever *message aggregation* occurs. Message aggregation is said to occur whenever more than one message is received on a single interrupt.

[7] This is not the latest or final build of Windows 2000.

[8] A least squares fit shows an R-squared value of 99%. Data not shown in this paper.

[9] A least squares fit shows an R-squared value of 99% also. Data not shown in this paper.

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

If we had a situation in which a message arrived every second each incoming message would most likely cause an interrupt. If however, we had a situation in which a second message came in while the first message was still being processed then the second message will not cause an interrupt. This is because interrupts are enabled only after the first message is *completely* processed. Hence we could wind up processing more than one message on a single interrupt. This is called message aggregation or coalescing. Of course, message aggregation can only occur between messages that are received on the same VI or messages that are received on the same CQ. If two messages arrive simultaneously at two completely independent VI's no aggregation can possibly occur. Hence one way to ensure a high message aggregation would be to receive messages using a small number of VI's or CQs. Message aggregation can be measured as the ratio of number of messages received to the number of interrupts. In a simple single thread ping-pong test the message aggregation, denoted by the symbol β=E[N], is 1. Assuming interrupts are not always turned ON, then the CPU cost per message, denoted by the symbol γ, would be

$$\gamma = (I_p + I_s + C_q + C_A + V) \qquad (4)$$

In a test which shows message aggregation the CPU cost per message would be

$$\gamma = ((I_p + I_s + C_q)/\beta + C_A + V) \qquad (5)$$

because only one out of every β messages causes an interrupt. From equation (5) we see that when β assumes large values (i.e greater than 10) the CPU cost becomes

$$\gamma \approx (C_A + V) \qquad (6)$$

A large value of β occurs when a large number of messages are received per second or whenever messages are received in bursts on a small number of VIs or CQs. In an application environment it may be difficult to restrict the number of VIs that are used to transfer messages so the only practical approach to achieve a high value for β would be to use a CQ. To demonstrate the effectiveness of message aggregation we ran a multi-threaded test to compare the Wait mode and the CQWaitD modes. The CQWaitD mode has the opportunity to benefit from message aggregation over a small number of CQs, namely 3 or 4. These results are presented in Table 4. From the CPU cost results in Table 4 we see that the CPU cost using the CQWaitD mode with 64 threads at a message size of 64 bytes is ≈ W µS. This corresponds to what we would expect using equation (3). The Wait mode also benefits from message aggregation in this scenario under heavy traffic conditions i.e with more than 4 threads. This is probably because of the way the application model behaves. The application model sends and receives

Table 3: This table shows the CPU cost components of the receive latency. The CPU cost of the send process is extremely small hence the CPU cost of a send-receive operation is defined entirely by the components in this table. The data in this table was obtained using software instrumentation in VI developed in-house

CPU Cost Component	Sym -bol	Time(µs)	Description
Interrupt Processing	I_p	8	CPU time spent in the ISR, the DPC, and the time between when the DPC is scheduled and when it starts running
Interrupt Switching	I_s	15	CPU time taken to turn interrupts on and off.
CQThread Context Switch	C_q	20	CPU time to context switch the CQThread into and off the CPU.
Application Thread Context Switch	C_A	20	CPU time to context switch the application thread into and off the CPU.
Test Overhead	V	5	CPU time spent in the VI test.

messages back-to-back. In a test with a large number of threads this results in a situation where we have messages being sent in a batch. These messages are then echoed in a batch so we have a burst of arrivals. If these arrivals are separated by an inter-arrival time that is less than the execution time of the interrupt service routine we are likely to see message aggregation in the Wait mode. In the CQWaitD mode, since there are only a small number of CQThreads each CQThread sees a higher message arrival rate than each VI would. In addition, since each
CQThread spends a user-specified duration on a message before checking for new arrivals, the inter-arrival time between successive messages to the same CQ can be higher without lowering the message aggregation rate. One approach to validate this explanation for the data shown in Table 4 would be to modify the application model so that each thread pauses or delays for some time between successive send-receive operations. This delay may in fact be viewed as an inter-arrival time between messages on a VI. We

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

plan to study the variation in CPU cost with different distributions for this inter-arrival time. In summary, the poll mode of completion wastes too much CPU time for transaction processing systems, as can be seen from Table 2. Hence the only reasonable modes of completion would be a wait, notify, CQWait or CQWaitD.

Table 4: Comparison of the CPU cost of the Wait and CQWaitD modes for varying number of application threads. In the CQWaitD mode all the presented data corresponds to tests with one CQThread and one CQ.

CPU Cost (μS)			
# Threads	CQWaitD	Wait	W=28 µs
4	W	W+12	
6	W	W	
12	W	W	
16	W	W	
20	W	W	
24	W	W	
32	W	W	
64	W	W	

The wait and CQWait modes are almost identical. The CQWaitD mode has a higher CPU cost than the Wait mode with low message aggregation but consistently lower CPU cost at high message rates. The notify mechanism [2] allows a user to name a function that will be run by VI using a notify thread when a message is received. This thread performs the following operations: context switches into the CPU, runs the function, picks up a message, wakes up the application and then context switches out of the CPU. Hence each message requires four context switches, two for the notify thread and two for the application thread. In the case of the CQWaitD mode with message aggregation the number of context switches can be reduced to a value close to two as shown in equation (6). Assuming that the work that need to be done once a message is dequeued is the same, the notify mode takes about two context switches more than the CQWaitD mode. Hence it can, at best, show as good performance as the CQWaitD mode.

6 Conclusions

In conclusion, we have shown that the CQ mechanism offers an optimal way (in terms of CPU cost) to receive messages in multi-threaded applications under moderate to heavy traffic loads. The analysis of CPU cost in this paper can be used to estimate an upper bound for the number of transactions that can be executed on a given system by an application. For a given application, if one measures the message arrival rate, and the average message inter-arrival time one can estimate a value for β using the latency data in Table 1. Given the data in Table 3, a value for β, and using equation (5) in section 5 one can estimate the CPU cost due to messaging. Then using equation (1) in section 2 one can estimate an upper bound on the number of transactions that can be executed on a given system.

7 Appendix A

Here we derive expressions for the probability of finding and empty queue in a M/G/1 system with different service time distributions for the customers arriving to empty versus non-empty queues. We will use the same terminology as in the body of the paper and so do not repeat it here. Further information about queues can be found in [6].

Suppose arrivals after time t>0 occur at time points A_1, A_2, ... and departures occur at time points D_1, D_2, ... and a departure at time D_i leaves behind a queue of size X_i (for i=0,1,..). Since the arrival process is Poisson, the sequence X_1, X_2,... is an embedded Markov chain (since the state X_{i+1} depends only on the state X_i, i>0). Let Z_n be the number of arrivals during the service time S_n that completes at time D_n.

$$X_{n+1} = \begin{cases} X_n - 1 + Z_n & \text{if } X_n > 0 \\ Z_n & \text{if } X_n = 0 \end{cases}$$

For the random variable Z_n, let $b_j = P(Z_n=j)$ and

$$G_B(z) = E[z^{Z_n}] = \sum_{j=0}^{\infty} b_j z^j$$

Given a service time of length x, we can write down an expression for the number of Poisson arrivals as follows.

$$P(Z_n = j \mid S_{n+1} = x) = \frac{(\lambda x)^j}{j!} e^{-\lambda x} \qquad j \geq 0$$

Integrating this by x, and applying the law of total probability, we obtain:

$$b_j = \int_0^{\infty} \frac{(\lambda x)^j}{j!} e^{-\lambda x} \frac{dF_B(x)}{dx} dx$$

Multiplying both sides by z^j and summing over all j, we obtain:

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.

$$G_B(z) = \int_0^\infty e^{-(1-z)\lambda x} \frac{dF_B(x)}{dx} dx = B^*(\lambda(1-z))$$

$$and \quad G_R(z) = R^*(\lambda(1-z))$$

Here the Laplace-Stieltjes transforms for the service times are defined as follows:

$$B^*(s) = \int_0^\infty e^{-st} dF_B(t)$$

$$and \qquad R^*(s) = \int_0^\infty e^{-st} dF_R(t)$$

We obtain a similar expression for the probability generating function for the number of arrivals during the first service time of a busy period. Let r_i be the probability of i arrivals during the first service time. The transition matrix for the embedded Markov chain $\{X_n, n>0\}$ contains elements b_j and the balance equation is as follows (where p_i is probability that i customers are left behind by a departing customer):

$$p_i = p_0 r_i + \sum_j p_{j+1} b_{i-j}$$

Now define the generating function for the steady state queue length probabilities as follows:

$$\Pi(z) = \sum_{i=0}^\infty p_i z^i$$

Applying the generating function to the balance equations now gives

$$\Pi(z) = \frac{p_0[G_B(z) - zG_R(z)]}{G_B(z) - z}$$

Since $\Pi(1)=1$, this simplifies to

$$p_0 = (1-\rho)/(1-\lambda m_R - \rho)$$

8 Appendix B

The mean response time for customers in this queue is simply the average queue length divided by the throughput.

$$E[W] = \frac{d\Pi(z)}{dz} / \lambda \qquad at \quad z = 1$$

The result is obtained after multiple applications of L'Hopital rule and considerable simplification.

$$E[W] = \frac{p_0(2m_R - \lambda^2(M_{2B}m_R + M_{2R}m_B) + \lambda(M_{2R} - 2m_R m_B))}{2(\lambda m_B - 1)^2}$$

References:

[1] D. Dunning et al, "The Virtual Interface Architecture", IEEE Micro Mar/Apr 1998, pp 66-75.

[2] "Virtual Interface Architecture Specification", revision 1.0.

[3] E. Speight et al, "Realizing the Performance Potential of the Virtual Interface Architecture", in Proc. International Conference on Supercomputing, 1999, pp. 184-192

[4] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: a user-level network interface for parallel and distributed computing", in Proc. ACM Symposium on Operating Systems Principles, 1995, pp. 40-53

[5] T. von Eicken, et al, "Active messages: a mechanism for integrated communication and computation", in Proc. of the Symposium on Computer Architecture, 1992, pp. 256 -266

[6] P.G. Harrison and N. M. Patel, "Performance Modelling of Communication Networks and Computer Architectures" Addison-Wesley 1992.

[7] G. Banga, Jeffrey C. Mogul, Peter Druschel, "A Scalable and Explicit Event Delivery Mechanism for UNIX" in Proc. of the USENIX Annual Technical Conference, June 1999

[8] Philip Buonadonna, et al, "Implementation and Analysis of the Virtual Interface Architecture", in Proc. of Supercomputing 1998, Orlando, Florida, Nov 7 - 13, 1998.

[9] C. Dubnicki, L. Iftode, E. W. Felten, Kai Li, "Software support for virtual memory-mapped communication", in Proc. of IPPS '96, Honolulu, April 1996. pp. 372-381.

[10] D. E. Culler, L. Tin Liu, R. P. Martin and C. Yoshikawa, Assessing Fast Network Interfaces, IEEE Micro, vol. 16, (no. 1), Feb 1996, pp. 35-43

[11] B. N. Chun, A. M. Mainwaring, D. E. Culler. Virtual network transport protocols for Myrinet, IEEE Micro, vol. 18, (no.1), Jan.-Feb. 1998, pp. 53-63

[12] "InfiniBand Architecture Specification", revision 0.9, March 31, 2000, http://www.infiniband.org.

[13] K.Langendoen, R. Bhoedjang, B. Henri, "Models for Asynchronous Message Handling" IEEE Concurrency, April-June 1997, pp. 28-38.

Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:55:35 UTC from IEEE Xplore. Restrictions apply.