Using Warp to Control Network Contention in Mermera*

Abdelsalam Heddaya
heddaya@cs.bu.edu

Kihong Park†
park@cs.bu.edu

Himanshu Sinha
hss@cs.bu.edu

BU-CS-93-007
June 4, 1993

Boston University
Computer Science Department
Boston, MA 02215
Phone: 617 353-8919
Fax: 617 353-6457

Abstract

Parallel computing on a distributed system, such as a network of workstations, can saturate the communication network, leading to excessive message delays and consequently poor application performance. Current operating systems offer only partial support for flow control protocols that can help insulate application performance from extraneous traffic on the shared network. We examine empirically the consequences of integrating one such protocol, called Warp control [Par93], into Mermera, a software shared memory system that supports parallel computing on distributed systems [HS93].

Preliminary performance measurements are reported for an asynchronous iterative program to solve a system of linear equations, under varying levels of network contention. The experiments were conducted on a network of seven Sun Sparc 1+ workstations, using an auxiliary traffic generator. These measurements show that Warp succeeds in stabilizing the network behavior when there is high contention, increasing the effective throughput available to the application, and consequently decreasing its completion time. In some cases, however, Warp control does not achieve the performance attainable by fixed size buffering when using a statically optimal buffer size. Based on the nature of Warp and the underlying communication layers, we offer explanations for our results.

Our use of Warp to regulate the allocation of network bandwidth emphasizes the possibility for integrating it with the allocation of other resources, such as CPU cycles and disk bandwidth, so as to optimize overall system throughtput, and enable fully-shared execution of parallel programs.

Keywords: Distributed non-coherent shared memory, network contention, flow control, iterative methods, Isis.

*This research was supported in part by NSF under grants IRI-8910195, IRI-9041581 and CDA-8920936.
†This author was additionally supported by NSF grant CCR-9204284.

1 Introduction

Today's high performance networked workstations are increasingly used for parallel computing. Such systems possess networks (*e.g.*, Ethernet, FDDI) that are proportionately much slower than parallel computer interconnection networks. Consequently, parallel programs can place very high sustained communication loads on workstation networks, causing congestion and extraordinarily high communication delays. This phenomenon can severely limit the number of workstations that can participate fruitfully in parallel when running communication intensive applications.

Mermera is an example of a system that can encourage applications to generate high traffic volumes, causing network congestion. It is a software shared memory system that has been developed to support parallel computing on a workstation network [HS93, Sin93]. Processes that comprise a parallel program communicate through read and write operations to the shared memory provided by Mermera. Several different memory behaviors are supported, including one coherent[1], and three non-coherent (Pipelined Random Access Memory [LS88], Slow Memory [HA90], and Local Consistency [HS93]).

A key property of non-coherent memory operations supported by Mermera is that they are "*asynchronous*" in that they do not wait for any communication to be performed before they return. This allows the application program to issue as many shared memory operations as the processing power of the CPU allows, potentially flooding the network with communication requests. By contrast, coherent write operations are inherently "synchronous" in that they do not return until some message is successfully sent and received. Applications such as asynchronous iterative algorithms [BT89], and oblivious computations [LS88], are thus capable of causing high and sustained network contention. Operating systems that aim to support parallel computing, must therefore offer mechanisms for controling network contention to a degree higher than is available today. To study the question, and try to formulate requirements for the likely candidate mechanisms for this purpose, we implemented, tested and measured the Warp control protocol under Mermera.

Warp control [Par93] is a distributed end-to-end flow control protocol which uses a time-stamp based estimator of network state to adaptively throttle arrival rates so as to optimize network

[1] Our definition of coherence, first advanced in [HS92] is equivalent to that of sequential consistency [Lam79].

throughput. It consists of a set of protocol suites, of which *rate adjustment protocol* (RAP) is the basic suite used in the experiments. The network estimator is called *warp*, and it uses time-stamps to estimate relative change in network delay.[2] In case of single server queues, $warp \approx \rho$ where ρ is the network utilization (*i.e.*, throughput). Warp control has the desirable property of being asymptotically stable, although for networks characterized by unimodal load–service rate functions[3], additional controls are needed to ensure stability. Warp is particularly suited for steady, high-bandwidth traffic, a communication pattern supported by non-coherent writes in Mermera. Although the ideal place for Warp to reside is in the network interface unit (*e.g.*, network layer in the ISO-OSI hierarchy or lower), in the current implementation it was applied at the level of Mermera which itself runs on top of Isis. Due to Isis' own flow control algorithm and Ethernet's exponential back-off mechanism, Warp was receiving highly filtered information, but nevertheless achieving improved performance when running a communication-intensive application. In order to harness the computational resources of large networks of workstations effectively, we believe that the network must be treated as an integral part of the distributed system, to be managed in tandem with other resources. We elaborate on this point in sections 3 and 7.

In the next two sections, we describe the essential details of Mermera and Warp. Section 4 specifies our experimental set-up, and section 5 provides basic measurements that characterize the communication underlying Mermera with, and without, Warp. Preliminary measurements that exemplify the application performance improvements obtainable with Warp are given and interpreted in section 6. Section 7 discusses our conclusions.

2 Mermera

Mermera supports several different memory behaviors, which includes:

Coherent Memory: All processes agree on the order of *all* writes. In other words, if a process

[2] Throughout the paper, we will often use Warp to refer to the protocol and warp (lower case) to denote to the quantity defined.

[3] One method for modeling network congestion is to assume a functional relationship between network load and service rate that has a unimodal shape [Par93].

observes[4] some writes in a certain order then no other process observes those writes in a different order. Coherent memory is equivalent to sequentially consistent memory [Lam79].

Pipelined RAM (PRAM): The order of all writes by the *same* process is respected by all processes, *i.e.*, if a process performs two writes, w_1 followed by w_2, then no process can read them in the reverse order. Writes by different processes may be interleaved in different orders by different processes.

Slow Memory: All writes by *the same process to the same location* are ordered by all processes in the order they were written. Writes to different locations by the same process may be ordered differently by different processes.

Our current implementations[5] support the above behaviors using full replication of the shared memory. The interface consists of one read operation, and several write operations, one for each type of memory behavior. A read operation returns the value stored in the local copy of shared memory, and hence takes the same time as an ordinary memory reference. This time is typically 0.1–1.0 microseconds. Each write operation causes Mermera to enqueue the new value for broadcast over the network, possibly together with previously enqueued values. A coherent write operation returns only after the message has been sent and received back by the writer.

Slow and Pipelined RAM write operations do not wait for any communication to complete; once their request for communication is recorded locally, the operation can return. We can say that these operations are asynchronous with respect to communication. Hence, computations that use *Slow_Write* or *PRAM_Write* can, in principle, submit new values for transmission at a sustained rate greater than the network capacity.

Writes are propagated by broadcasting the values to other processes. We use the Isis toolkit [BJ87, BSS91] for our implementation because it provides a suite of multicast protocols that satisfy different ordering properties. The broadcasts of interest to us are *abcast*(), *fbcast*() and *mbcast*(). These primitives have different constraints on the order in which the messages are delivered to their destinations.

[4] A write $x.w$ to location x is said to be observed by process i if the value returned by a read by process i is the value written by $x.w$ or is a value that may have been influenced by $x.w$.

[5] Mermera has been implemented on the BBN Butterfly, on a workstation network, and on the CM-5.

All messages sent using *abcast*() are delivered in the same order at all destinations, *i.e.*, the order in which these messages are delivered is the same for all processes. This is exactly the property we want for *CO_Write*. The *fbcast*() messages obey a weaker constraint: messages sent by the *same* process are delivered to all processes in the order they were sent. However, *fbcasts* sent by different processes may be interleaved in different orders at different recipients. This suffices for *PRAM_Writes*, so we use *fbcasts* to propagate them. No ordering constraints are guaranteed among *mbcast*() messages[6]. We use this primitive for propagating *Slow_Writes*.

Isis offers no ordering guarantees for messages sent using different primitives, *e.g.*, if two messages are sent one after the other using *abcast*() and *fbcast*(), respectively, they are not necessarily delivered in the order they were sent. Our implementation enforces this ordering explicitly via sequence numbers, and by careful control of the order in which received updates are applied.

Isis employs the UDP communication protocol, which, in turn, runs on top of the Ethernet's CSMA/CD protocol. Both Isis and the Ethernet, but not UDP, have built-in flow control whose effect can be discerned in our measurements presented in sections 5 and 6.

Numerous software shared memory systems have been proposed in the literature, but most of them provide coherence at the level of sequential consistency or stricter. Recently, some systems started to allow weaker conditions. Examples include the DASH multiprocessor [LLJ+93], the Munin software shared memory system [CBZ91], and transactional memory [HM92], which allow sequential consistency to be broken, but only during the execution of code blocks explicitly marked by the programmer. These systems, as well as Mermera, stand to benefit from network contention control protocols.

3 Warp Control

Warp control is a distributed end-to-end flow control protocol that uses a time-stamp based scheme to throttle arrival rates at active message sources to achieve optimal network utilization. For a related discussion of flow control algorithms, see [GK80, HW91]. Let N be the network, and let $p_1, p_2, \ldots, p_n$ be the network nodes. Let μ be the service rate of the network, and let $\lambda_1, \lambda_2, \ldots, \lambda_n$

[6] The current implementation of Isis does deliver them in the order they were sent by the sender.

be the arrival rates of $p_1, p_2, \ldots, p_n$, respectively. Thus the total arrival rate of the network is given by $\lambda = \sum_{i=1}^{n} \lambda_i$. Let L denote the load of N, and let $\rho = \mu/\lambda$ be the utilization. For simplicity of exposition, assume that every node p_i has access to the same global clock[7]. Assume at time t_1, node p_j wants to send a packet m_{ij} to node p_i. The following protocol is executed at the sender p_j:

Message encoding protocol (MEP):

1. Create a header containing i, j, and time stamped with t_1.

   Denote the latter operation by $m_{ij}.time\_stamp := t_1$.
2. Attach the header to the data section of m_{ij} and submit to network N.

Let t_2 be the time at which m_{ij} arrives at p_i. For all $k \in \{1, 2, \ldots, n\} \backslash \{i\}$, p_i maintains a data structure $hist[k]$ containing two fields $hist[k].last\_in$ and $hist[k].last\_out$. In $hist[k].last\_in$ is recorded the time at which the last message from p_k arrived at p_i, and $hist[k].last\_out$ records the time at which it was sent out from p_k. At time t_2, the following protocol is executed at the receiving end p_i:

Message decoding protocol (MDP):

1. $warp := (t_2 - hist[j].last\_in) \,/\, (m_{ij}.time\_stamp - hist[j].last\_out)$.
2. $hist[j].last\_in := t_2$.
3. $hist[j].last\_out := m_{ij}.time\_stamp$.

It can be shown [Par93], under the assumption of N being a FCFS single-server queue and $|d\lambda/dt| \ll 1$, that $warp \approx \rho$. Therefore, to achieve and maintain maximum utilization, the following control law can be employed,

$$\frac{d\lambda}{dt} = \epsilon(1 - warp) \tag{1}$$

where $\epsilon > 0$ is a constant factor. If τ is the network delay, one can further show that the resulting system is asymptotically stable if

$$0 < \tau\frac{\epsilon}{\bar{\mu}} < \pi/2 \tag{2}$$

[7]In general, this assumption can be relaxed so that, not only do the network nodes need no access to the same absolute time, but their local clocks may also run at different speeds.

where $\bar{\mu}$ is the mean service rate. Thus to attain maximum utilization in a stable manner, it suffices to set ϵ very small. In practical terms, this translates to an initial transient period at start-up time, and a lagged response to sudden structural changes (*e.g.*, 30% of the nodes going down at once). On the positive side, this makes the system robust with respect to noise, guaranteeing asymptotic stability even under noisy conditions. The issues of better responsiveness, fairness and stability under unimodal load-service rate functions are discussed in [Par93].

Distributed control is achieved by noting that $\lambda = \sum_{i=1}^{n} \lambda_i$. That is, at every p_i, the following update protocol is executed after performing MDP:

Rate adjustment protocol (RAP):

1. $\lambda_i(t) := \lambda_i(t-1) + \epsilon_i(1 - warp)$.
2. If $\lambda_i(t) < 0$, then $\lambda_i(t) := 0$.

A straightforward way of implementing message submission rate control at every p_i using a *buffering scheme* is to maintain a large message queue B, and enqueue message submissions to B. The network interface unit at p_i periodically examines B, and at every such instant dequeues $\lambda_i b$ messages from B. It executes MEP on the dequeued messages and sends them off. The parameter $b > 0$ is system dependent, and affects the quantization range of λ_i. Information regarding the current state of B and $\lambda_i b$ can be made available to the application running at p_i, so that it can better manage its communication needs should its computation allow it to do so. For example, in asynchronous iterative algorithms that solve fixed-point problems arising in various applications [BT89], convergence rate is enhanced if the application can keep its computation/communication ratio, which is variable, compatible with $\lambda_i b$. This way, no time is wasted in generating messages that may get delayed due to unnecessary buffering. For applications that do not have controllable computation/communication ratios, this benefit cannot be harnessed.

Nevertheless, from an operating systems point of view, if the operating system is managing a set of processes each with its own nonvariable computation/communication ratio, the operating system itself may schedule the processes so that the system as a whole stays as compatible as possible with $\lambda_i b$. For example, if $\lambda_i b$ is small, then it is wasteful for the operating system to allocate CPU cycles to processes having a small computation/communication ratio (*i.e.*, communication intensive) if it has a choice. Not only will the delay be large due to the growing queue, but in the extreme case when B is full, CPU cycles will be wasted trying to write to a buffer that is already full. These speculative scenarios illustrate the close relationship between various resources such as computation (CPU) and communication (network) in networked workstations that share them. Thus, the possibility clearly exists for an operating system to enhance system performance by allocating CPU and network resources in tandem.

In the current implementation, due to issues specific to the present design of Mermera, a simplified scheme is being used. A buffer B is maintained between Mermera and Isis, and if $|B| > \Lambda_i b$, where $|B|$ denotes the queue length, the buffer is flushed to Isis. Λ_i is a quantity that has the opposite effect of λ_i, in the sense that, if Λ_i increases, the message submission rate decreases due to less frequent flushing. This modified implementation is easily accommodated by changing the sign of ϵ in equation (1) from positive to negative. The value referred to as "lambda" in this paper is actually Λ, and should not be confused with the arrival rate λ.

4 Experimental Set-up

We conducted our measurements on a network of six dedicated Sun Sparc 1+ workstations and a server, each equipped with 660 MByte individual disks, and connected via a private 10 Mbit Ethernet. The workstations all run SunOS version 4.1.1, and the Isis Toolkit version 2.2.5. We ran NetMetrix version 3.0.1, on the server and one of the workstations, to generate background traffic in order to increase contention for the Ethernet.[8] The remaining five workstations were dedicated to running the parallel equation solver on top of Mermera and Isis.

We measured the completion time of the equation solver on a system of linear equations $Ax+b = 0$ in 1000 variables, generated pseudo-randomly. The level of traffic infused by NetMetrix on each of the two machines, given as an uninterpreted percentage, was varied between 0% to 65%. We also measured the input and output quantities for Warp control, *warp* and Λ, defined in the previous section.

Warp was implemented as a function call from Mermera, invoked each time a message is received.

[8] We are grateful to Metrix Network Systems, Inc., for free use of their software.

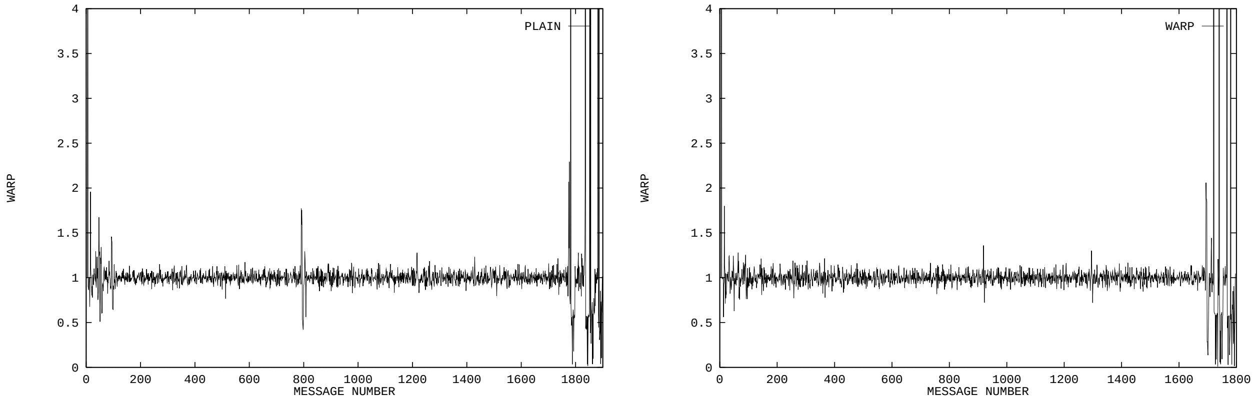

Figure 1: In the absence of background traffic, *warp* values indicate that the network is well behaved both without Warp (left) or with it (right).

The corresponding message rate control is incorporated as a conditional check, tested at every message enqueue, flushing the buffer to Isis if $|B| > \Lambda_i b$. This places Warp control at an inefficiently high level, acting on data that is filtered through lower layer protocols. For operating system support purposes, Warp should be moved to the transport/network layer of the ISO-OSI reference model.

5 Network Performance

Figure 1 shows the variations in network delay (as measured by *warp*) during a typical run of the equation solver on top of Mermera. With only five workstations, the network is clearly operating well within its capacity, and therefore is very well behaved. Turning on Warp control does not lead to any noticeable performance improvement. In order to see what might happen if the network contention was high, for example, caused by having 100 workstations instead of only five, we used a traffic generation program that is part of the NetMetrix software. We ran the traffic generator on two dedicated workstations, other than the five that run the equation solver, and obtained the results shown in figure 2.

Compared to figure 1, *warp* values in figure 2 are far higher, with many spikes exceeding 5, some exceeding 50, and a few exceeding 500. A value of 5 means that a Mermera internal message took five times longer to reach its destination than the immediately preceding message from the same

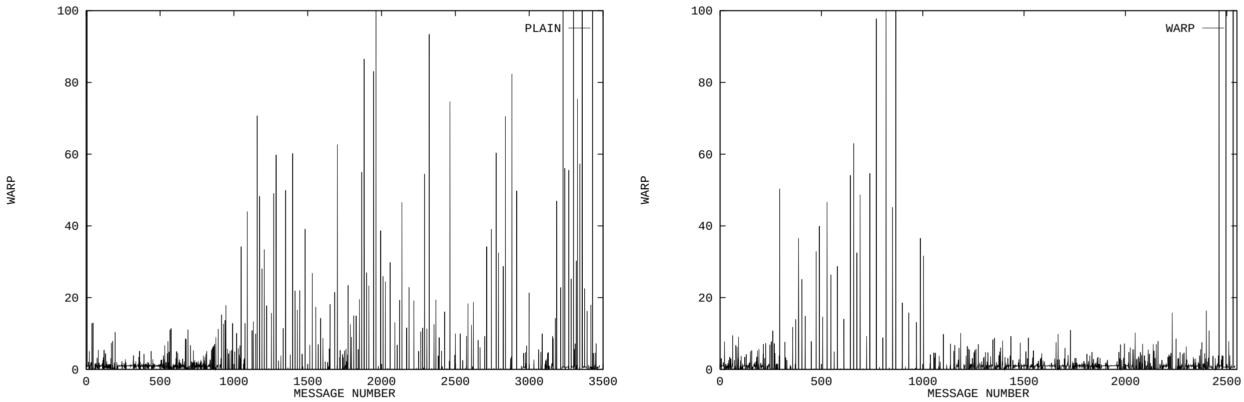

Figure 2: At 65% background traffic, *warp* values exhibit numerous high spikes (above 10) without Warp control (left). Warp control (right) effectively eliminates them, once it stabilizes. Spikes above 100 are clipped to preserve scale.

source to the same destination. Since Mermera messages, which are sent via Isis, take roughly 10-50 ms on average to reach their destination, our measurements suggest that some messages are taking 0.5-2.5 seconds—an extraordinarily long delay. Such large differences in message delay can be explained by looking at the flow control mechanisms used by Isis and the underlying Ethernet. Isis uses several schemes, among them a simple heuristic that, based on the number of buffered messages at a node, determines whether congestion exists or not. Once Isis decides that congestion exists, it stops sending messages for a long period of time (1 second) [Bir93], thus causing a large spike in *warp*. Smaller spikes may be explained by the Ethernet's back-off mechanism when collisions occur [Tan88].

Figure 2 offers an empirical insight into the behavior of Warp: it slowly but effectively squelches large spikes in *warp*, and consequently in message delay. We see in the right hand plot that the spikes subside after a thousand messages or so have been delivered. Warp's success in smoothing the network's behavior reduces the total number of messages needed to run the application to completion, from ~ 3500 to ~ 2500. Both of these observations are further elaborated in the next section.

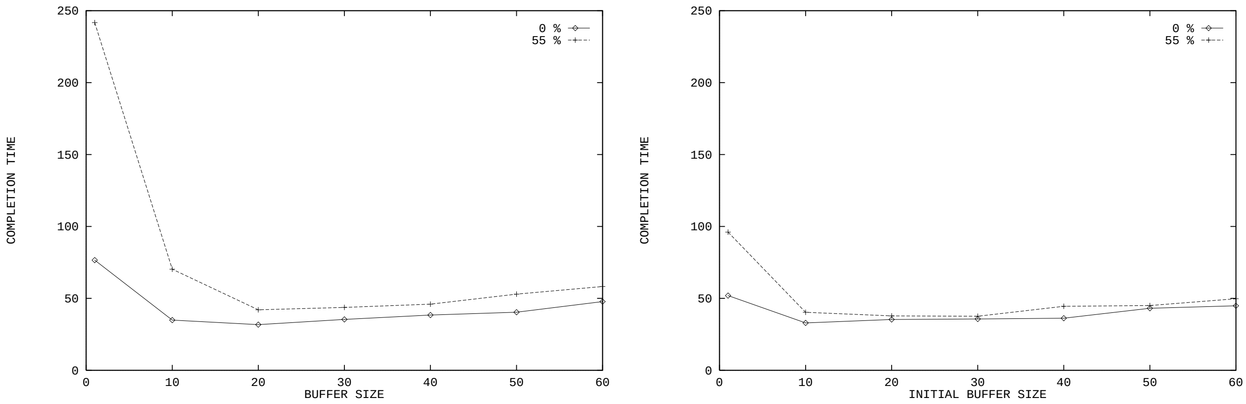

Figure 3: Warp (right) reduces the sensitivity of completion time to background traffic level, compared to the case without Warp (left).

6 Application Performance

Our preliminary measurements suggest that Warp control can improve application performance by enabling it to extract a higher communication throughput, with a correspondingly smaller delay, from the network. In this section we present evidence to support this conclusion, and empirically analyze the behavior of Warp control in order to explain the precise nature of the performance improvement it affords. In particular, we seek to illuminate the relationship between performance under statically optimal buffer size, and that under dynamically optimal buffer size, controlled by the Warp protocol.

Consider the two different views of the completion time versus the buffer size (initial buffer size when Warp is active), found in figures 3 and 4. Figure 3-left illustrates the detrimental effect network contention can have on the equation solver's completion time, which is manifested as a shift in the completion time curve. When there is high background traffic and Warp is turned off, simply increasing the fixed buffer size causes a reduction in network contention that is sufficient to decrease the completion time drastically (from about 250 seconds to less than 50 seconds). The vertical separation between the two curves in the same plot (figure 3-left) indicates the impact of background traffic on completion time, which can also be very high when the solver is itself generating high traffic (small buffer size). Warp, when activated, reduces the sensitivity of completion time to the amount of background traffic that exists on the network, suggesting that Warp helps the equation

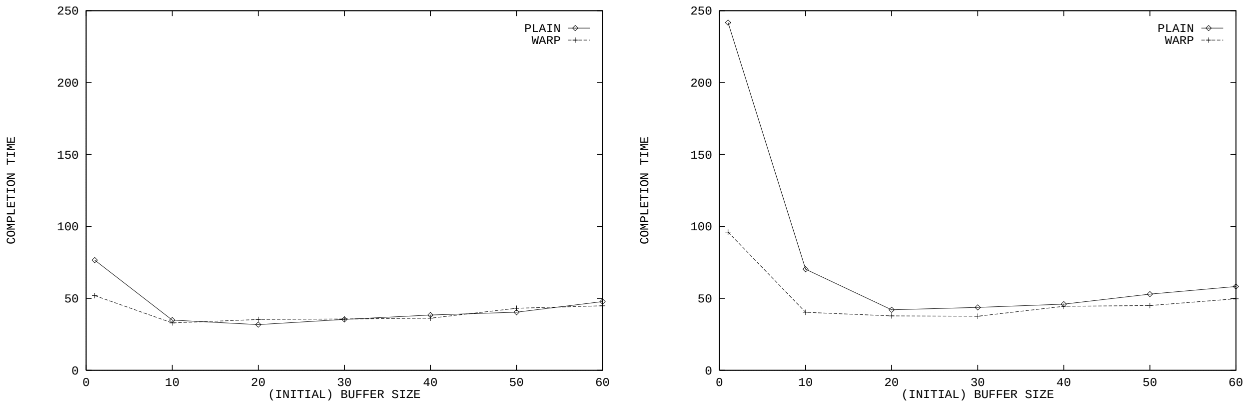

Figure 4: Completion time with five compute processors and two traffic generation processors with and without Warp control. Left and right graphs reflect 0% and 55% background traffic levels, respectively.

solver extract sufficient throughput from the network even in the presence of 55% background traffic. That is, Warp is able to insulate the system, to a significant degree, from the otherwise harmful effects of increased traffic.

Figure 4 allows us to quantify the performance gains obtained with Warp. The graphs are self-explanatory, showing that Warp can relieve the application programmer, or Mermera's designer, from the need to choose a buffer size appropriate to the particular level of contention that happens to be in effect on the network. Furthermore, in an ever changing environment, Warp provides a simple means for effective adaptation. This suggests the need for operating system tools supportive of such a mechanism, to insulate the programmer from system-dependent communication effects.

However, a very revealing measurement is not apparent in these graphs. Consider the case for an (initial) buffer size of 10, and background traffic level of 65% (only slightly different from the 55% case shown in figure 4-right). The no-Warp run exchanges 34,436 messages, and completes in 95.2 seconds, for a communication throughput of **3,625** updates/second. By contrast, the run under Warp transmits 27,435 messages, and finishes in 52.6 seconds, achieving the *much higher throughput* of **5,276** updates/second. In effect, by reducing the variance in network delay, via controlling excessive traffic, effective throughput is increased, which leads to faster overall performance.

One obvious remaining question is why doesn't Warp control meet or even beat the best completion time, when it starts with an initial buffer size that is far from optimal? To answer this

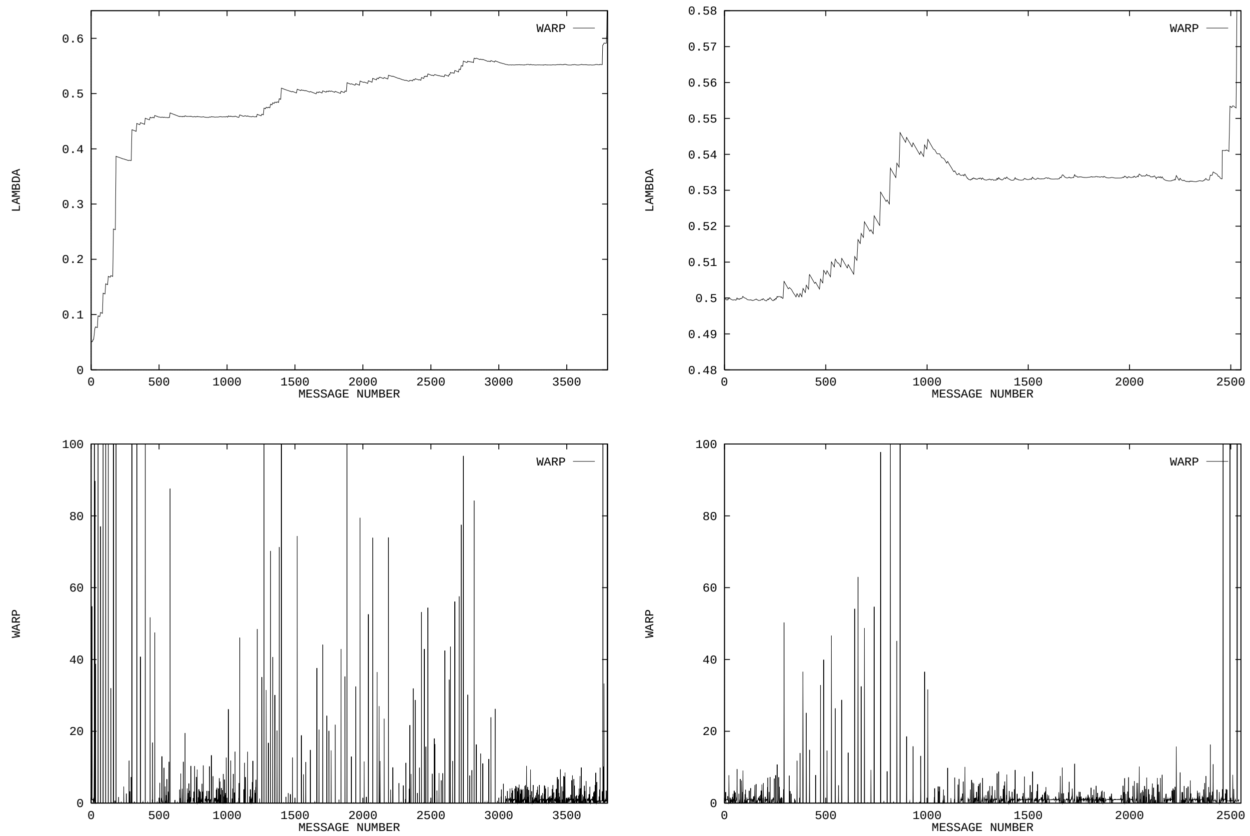

Figure 5: Λ (upper graphs) and *warp* (lower graphs) for background traffic level of 55% with initial $\Lambda = 0.05$ (left graphs); and traffic 65% with initial $\Lambda = 0.5$ (right graphs). Buffer size used for each message sent is 20Λ, hence the smallest effective change in Λ is 0.05.

question, and offer additional insight into the network dynamics under Warp, we present the results in figure 5, which shows the actual buffer sizes computed by Warp control throughout two executions. The left hand side pair of graphs correspond to an initial buffer size of 1, while the right hand side pair depict the case for initial buffer size of 10. Clearly, Warp control stabilizes much more slowly when Λ starts farther from its eventual operating neighborhood. In the first case, it takes about 3000 messages for Warp to stabilize, while in the second case, it does so after only 1000 messages. The general slowness with which Warp reacts has the advantage of ensuring its long term stability, but, for applications running for short durations, the initial transient period may not get sufficiently amortized to yield significant performance enhancement.

7 Discussion

This paper deals with a central phenomenon that was brought to our attention through experience with parallel applications running on top of Mermera, namely, extreme changes in message delay and degradation in communication throughput as a result of network contention. Clearly, this phenomenon should not be left to application programmers, or even presentation layer developers, to contend with. For example, neither the equation solver programmer, nor the Mermera designer, have the necessary information to be able dynamically to adjust their software's communication requirements to the levels that would optimize network performance in a changing shared environment. Unfortunately, in the absence of specific operating system support for a flow control protocol such as Warp at lower layers of the OSI-ISO hierarchy, these programmers are left with no other alternative but paintstakingly tune their applications by hand, *and* live with performance that can be optimal only under very restricted circumstances, if at all.

We have shown that in systems like Mermera, with a pronounced need for flow control as a result of the "asynchrony" of their communication, that Warp can help smooth network behavior, and deliver enhanced performance to applications. Our measurements also give experimental insight into the dynamic behavior of Warp itself, confirming the theoretical predictions and large-scale simulations of [Par93], given for a general packet or circuit-switched queueing system. Furthermore, our data revealed to us an undocumented aspect of the Isis toolkit, which is the technique it uses to control congestion. It was intructive for us to note that in every run, Warp was able eventually to smooth network traffic enough to prevent Isis' own mechanism from kicking in.

One can think of Warp control as a bandwidth allocation mechanism. This raises many questions, briefly touched upon in section 3, including the possibility of tying CPU scheduling to network bandwidth allocation in order to improve total system throughput. Such a "wholistic" perspective has obvious benefits for applications other than parallel computing on distributed systems. For example, multimedia applications could benefit from coordinated scheduling of disk I/O, CPU cycles, and network bandwidth.

References

[Bir93] Kenneth Birman. Congestion control in Isis. Private communication, May 1993.

[BJ87] K. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. 11th ACM Symp. on Operating System Principles, Austin, Texas*, Nov. 1987.

[BSS91] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, Aug. 1991.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenopoel. Implementation and performance of Munin. In *Proc. 13th ACM Symp. on Operating System Principles, Asilomar, California*, pages 152–164, Oct. 1991.

[GK80] M. Gerla and L. Kleinrock. Flow control: a comparative survey. *IEEE Trans. Comm.*, COM-28:553–574, 1980.

[HA90] P.W. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Proc. 10th IEEE Intl. Conference on Distributed Computing Systems, Paris, France*, June 1990. Also available as Georgia Institute of Technology technical report GIT-ICS-89/39.

[HM92] M. Herlihy and J.E.B. Moss. Transactional Memory: architectural support for lock-free data structures. Technical Report CRL 92/07, Digital Equipment Corp., Cambridge Research Lab, Dec. 1992.

[HS92] Abdelsalam Heddaya and Himanshu S. Sinha. Coherence, non-coherence and Local Consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Boston University, Computer Sciene Dept., May 1992.

[HS93] Abdelsalam Heddaya and Himanshu S. Sinha. An overview of MERMERA: a system and formalism for non-coherent distributed parallel memory. In *Proc. 26th Hawaii International Conference on System Sciences, Maui, Hawaii*, Jan. 5–8 1993.

[HW91] Z. Haas and J. Winters. Congestion control by adaptive admission. In *Proc. IEEE INFOCOM*, pages 560–569, 1991.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sep. 1979.

[LLJ+93] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: logic overhead and performance. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):41–61, Jan. 1993.

[LS88] R.J. Lipton and J.S. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR-180-88, Princeton Univ., Dept. of Computer Science, Sep. 1988.

[Par93] Kihong Park. Warp control: a dynamically stable congestion protocol and its analysis. In *Proc. ACM SIGCOMM*, 1993. To appear.

[Sin93] Himanshu Sinha. MERMERA*: Non-coherent Distributed Shared Memory for Parallel Computing.* PhD thesis, Boston University, Computer Science Department, 111 Cummington Street, Boston, MA 02215, May 1993.

[Tan88] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition edition, 1988.