HETEROGENEOUS CHANNEL BONDING REVISITED

Daniel Andresen
Department of Computing and Information Sciences
Kansas State University
email: dan@cis.ksu.edu

Sterling Hanenkamp
Department of Computing and Information Sciences
Kansas State University
email: sterling@cis.ksu.edu

ABSTRACT
Efficient communication in distributed systems is essential for optimal system performance. As parallel computation develops, many different kinds of messages need to be sent between machines. These messages may have different requirements for network latency and bandwidth. In this paper, we revisit combining two physical interfaces into one virtual network (channel bonding).In contrast with most existing implementations, we communicate via *heterogenous* interfaces, and explore a number of more sophisticated scheduling algorithms than the round-robin technique used in the standard Beowulf channel-bonding driver. We explore the relative tradeoffs of interface scheduling overhead versus the additional bandwidth possible with multiple interfaces. Using both Gigabit Ethernet and Fast Ethernet Networks, we evaluate system performance using netPerf. We show significant performance gains for small messages, with incremental improvements for larger packets.

KEY WORDS
Network performance, Linux, heterogeneous channel bonding, Beowulf.

1 Introduction

The Beowulf parallel system defines a new operating point in price-performance for single-user computing systems. It couples the low cost, moderate performance of commodity personal-computer subsystems with the emergence of standards in message passing hardware and software to realize an environment with exceptional local file storage capacity and bandwidth. This system was originally created and motivated by requirements of NASA earth and space science applications including data assimilation, data set browsing and visualization, and simulation of natural physical systems. It exploits parallelism in processor, disk, and internal communication, all derived from mass market commodity elements. This enables large temporary data sets to be buffered on the workstation in order to reduce demand on shared central file servers and networks while greatly improving user response time. The Center of Excellence in Space Data and Information Sciences (CESDIS) of NASA Goddard Space Flight Center has initially built the Beowulf architecture [9]. These systems significantly benefit bulk computations by distributing computation among machines in an off-the-shelf PC machine cluster.

While most distributed computing systems provide general purpose multiuser environments, the Beowulf distributed computing system is specifically designed for single user workloads typical of high end scientific workstation environments. These applications typically require high-bandwidth, low-latency communication. One approach is to design expensive custom hardware, such as Myrinet or the Princeton Shrimp project [3, 2]. Beowulf systems, in contrast, often incorporate no special purpose parts, depending instead on parallel Ethernet communication channels (channel bonding) to achieve adequate sustained interprocessor message transfer rates [4, 5, 7, 1]. This has required some software enhancements at the operating system kernel level, but has been achieved with commercial off-the-shelf hardware elements, specifically low cost Ethernet cards.

Previous research efforts have characterized heterogenous channel bonding. In [4], Kim and Lilja compared channels with significantly differing speeds (10Mbps Ethernet and HIPPI), and concluded the most efficient algorithm was to send small messages to the slow channel (since they were latency-limited), and large messages through the fast channel, to take advantage of the higher bandwidth. However, this strategy is suboptimal in the case of large numbers of small messages. Zhao and Andresen in [1] evaluated a 100Base-T and 1000Base-F Ethernet network, and determined that a simple N-to-1 strategy was optimal in that case. In the years since these papers were published, however, processor speeds have gone up exponentially,and homogenous channel bonding (popular at the Fast Ethernet level) has dropped in popularity with the availability of cheap gigabit adapters. The general attitude has been, “Why bother?”

In this paper we revisit the question of whether it is worthwhile, in today’s environment, to perform heterogenuous channel bonding. We present and evaluate the performance of a number of different packet scheduling algorithms. These reside at the virtual device level within the Linux kernel, and provide differing methods for determining which physical device a packet should be sent through. The Linux operating system kernel has been modified to dynamically distribute message packet traffic to load balance across both networks.

In section 2, we briefly discuss the standard Linux networking architecture and channel bonding. We then move on to present the packet scheduling algorithms and their implementation in section 3, and the experimental results in section 4. Finally, we conclude with a brief summary of our findings and future work in section 5.

2 Background: Linux networking

There are several different communication networks, such as Ethernet, ATM, Fiber Channel, HiPPI and FDDI, that have been proposed to be used as a communication channel between independent processing nodes. Each of these networks has different latency and bandwidth characteristics. For balancing the transmission load between heterogeneous networks in a parallel application we need not only match each message to the most appropriate network, we also need to coordinate multiplexing each message to two channels and combining them together.

In Linux operating system the network layer tries to be fairly objected-oriented in its design, as indeed is much of the Linux kernel [8, 6]. Every layer (Figure 1) treats an incoming packet as an object, and adds its own header to the packet. In order to let two physical network interface communicate between machines simultaneously, we need to set two interfaces with the same IP address, same Netmask, same MTU and the same hardware address. When data arrives from the IP layer, the device controller will pick up the packets from the packet queue and send them to different interface. On the other hand, whenever any interface has a message coming in from the physical layer it will interrupt the system interrupt handler and the put the packet into the receiving queue for upper layer processing. The key objects which we use to implement our virtual network interface are as following:

- **Device or Interface:** A network interface is used to send and receive packets. Usually, this involves both hardware and software. However, some devices are software only such as the loopback device which is used for sending data up the same stack. Device structure is the networks working structure. It contains all the information for both sending and receiving. It also has a slave field for supporting multiple interfaces, we will utilize this field to build our channel bonding.

- **Protocol:** Each protocol is effectively a different language of networking. Some protocols exist purely because vendors chose to use proprietary networking schemes, others are designed for special purposes. Within the Linux kernel, each protocol is a separate module of code which provides services to the socket layer.

- **Socket:** The name socket is derived from the notion of plugs and sockets. A socket is an endpoint in the network that provides unix file I/O and is accessed by an application through a file descriptor. In the kernel each socket is represented by a pair of data.

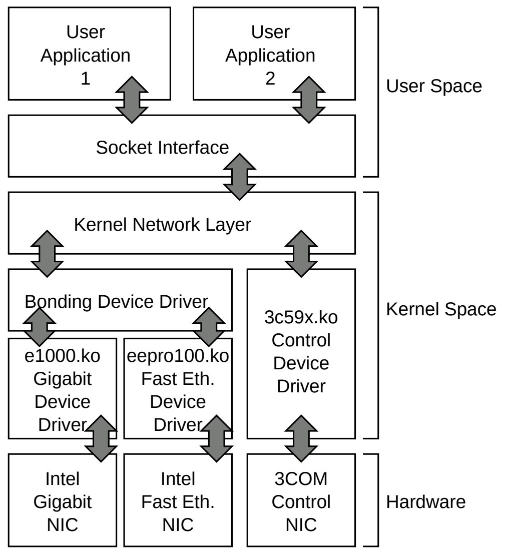

Figure 1. Linux networking configuration.

3 An enhanced channel bonding architecture

In general, channel bonding can be performed between n machines over m physical devices. We limited this experiment to exactly two machines with two devices each. We then experimented with a number of bonding modes. The first bonding mode we considered was one that is already available within the kernel. The others used, we designed ourselves–though, the implementation was generally just a variation on the existing mode used. The bonding modes used during the experiment include the following:

- **Round-Robin:** The bonding mode available with the kernel is named "Round-Robin." It is present in the stock kernel was the basis for the modes of our own design. The Round-Robin driver essential routes packets to every bonded interface. In our experiment, this meant alternating between routing packets through the Gigabit interface and through the Fast Ethernet interface.

- **Null:** The simplest bonding mode we designed we call the "Null" mode. In Null mode, there is no actual bonding at all. The Null mode driver, wraps a single interface with a bonding interface that performs the locking and other activities required at a minimum to wrap one network driver in another. The goal of this mode is to ascertain what differences in performance exist between the hardware network driver alone and using the hardware driver wrapped inside of the bonding driver. This would give us a base-line by which all other bonding drivers could be judged.

- **N:1:** Logically, since one interface we are testing is ten times faster than the other, we can write a driver mode which would perform the same function as Round-Robin, but using a 10:1 ratio. We generalized this driver mode as "N:1." This allows us to experiment with different ratios to compare their relative utility.

- **Least-Queue:** Moving beyond a simple ratio, we decided to try and take advantage of the queue length measurements recorded by the kernel for hardware devices. Armed with this information, we can try to direct packets to the device that is least busy at the current moment. Since our architecture is intended for use with devices with differing bandwidth we will always choose the faster interface if the queue length is the same on all devices. We refer to this driver mode as "Least-Queue."

- **N:1/Primary:** Then, our most complex driver modes are hybrid algorithms that try to use different techniques based upon packet sizes. The rationale being that a large packet is more likely to benefit from transmission over a high bandwidth interface. On the other hand, except for extremely small packets, it conceptually matters very little which interface smaller packets are sent through. Our first complex driver mode is called "N:1/Primary Combo." For packets below a certain threshold we transmit the packets according to the N:1 strategy while larger packets will always be sent out of the *primary* (i.e., higher bandwidth) interface.

- **Cut-off-Least-Queue:** We call the second complex driver mode, "Cut-off-Least-Queue." In this case, large packets are always transmitted to the primary interface. Small packets, then, will be transmitted according to the Least-Queue algorithm. This algorithm should conceptually do well where there is a high volume of small packet traffic that will split the traffic best among any interface. When a large packet does come through, though, we can always use the primary interface for faster transmission.

4 Experimental Results

In this section, we evaluate the algorithms presented in Section 3 for latency, bandwidth, and CPU utilization. During these experiments, we used two identical Dual AMD Athlon MP PR1800 machines with three network interfaces. The Tyan K7X Pro motherboard in these machines has two integreted network interfaces. We used these interfaces for the experiment: an Intel 82545EM 10/100/1000Mbps controller and an Intel 82551QM 10/100Mpbs controller. We installed a third network interface into both machines to allow for remote communication with the machines during testing and for automation of the test process. For our enhanced channel bonding architecture, we used a stock Linux kernel for our experiments. All tests involved a stock 2.5.x kernel with some slight modifications by us in the channel bonding driver. All tests presented here were performed on a 2.5.66 kernel with our modifications added. We used the kernel modules available in the stock kernel to drive the hardware interfaces.

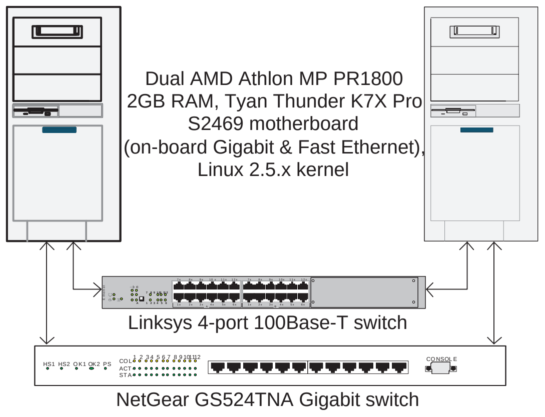
Dual AMD Athlon MP PR1800
2GB RAM, Tyan Thunder K7X Pro
S2469 motherboard
(on-board Gigabit & Fast Ethernet),
Linux 2.5.x kernel
Linksys 4-port 100Base-T switch
NetGear GS524TNA Gigabit switch

Figure 2. Experimental configuration.

Because channel bonding causes all interfaces to appear as one (including IP and MAC address configuration), it is necessary to use a separate network bus for each group of devices. For this experiment, we used a 4-port Linksys hub for routing fast ethernet traffic and a Netgear Gigabit Switch for the gigabit traffic. See Figure 2 for a view of the hardware setup.

These results were obtained through the Netperf network performance tester using it's `TCP_STREAM` test. All tests were done with the socket send and receive buffers set to 128 kilobytes (under Linux, this actually results in the maximum buffer size of 256 kilobytes). Each bandwidth measure was obtained by sending data over the network as quickly as possible for a 10 second period using a given message size. The tests were repeated over 20 iterations and the average of those results is presented here. We have measured the 95% statistical confidence interval for each point. The average error interval has been measured to be less than 10% of the bandwidth at each point.

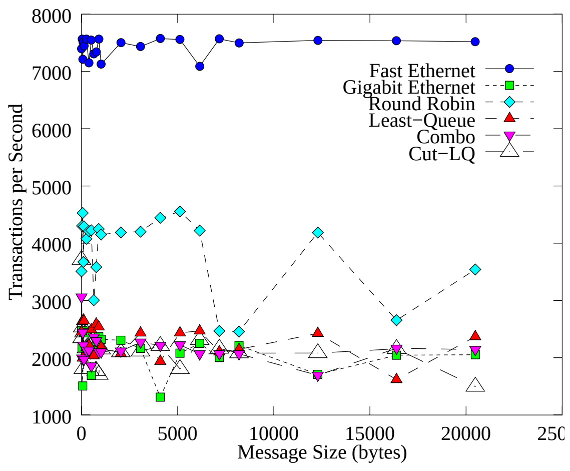

Figure 3. Messages/second (higher is better).

Latency: In general, Figure 3 indicates that plain Fast Ethernet has the lowest latency and highest number of messages per second, with Round-robin coming in second, and the rest of the algorithms clustered near each other at the bottom. Round-robin does show a surprising (but consistent) degree of variance, possibly due to CPU cache effects. The gigabit driver exhibits consistently more overhead than Fast Ethernet, with the more sophisticated algorithms giving nearly identical performance.

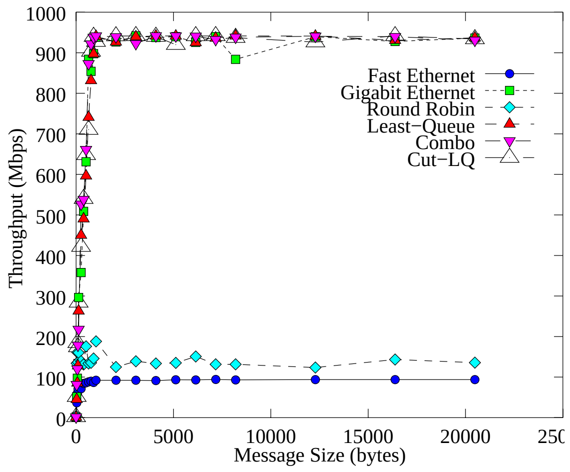

Figure 4. Throughput for all algorithms.

Bandwidth: We ran a number of tests on each of the bonding modes described in Section 3. As it turns out, not many of these driver modes proved to be a significant improvement over gigabit alone. For very large message sizes, no routing algorithm we present here has demonstrated notable improvement over gigabit alone. For very small message sizes, however, we have found channel bonding may provide a large bandwidth boost. Using the Round-Robin mode that exists in the stock 2.5.x kernel, small messages transmitted over TCP can gain as much as a 300% bandwidth boost over either fast ethernet or gigabit ethernet alone. The gain is as high as 150% above the sum of the bandwidth of both gigabit and fast ethernet interfaces.

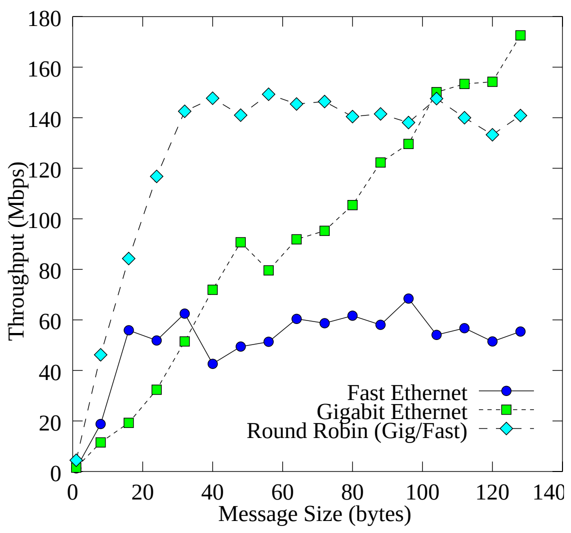

Figure 5. Fast Ethernet and Gigabit Ethernet results compared to Round-Robin Bonding.

As can be seen in Figure 5 (the interesting part of Figure 4), the results show that Round-Robin mode maintains a superior bandwidth for messages smaller than 64 bytes. On the test architecture, Round-Robin quickly becomes inferior as message sizes larger than 64 bytes as the gigabit device ramps up well-above the maximum bandwidth Round-Robin is of which capable. The reason for Round-Robin's impressive success appears to be that it reaches its bandwidth cap faster than the hardware devices do on their own. Since its bandwidth cap is so low, however, it cannot win once hardware processing of packet data trumps CPU processing.

A detailed examination of the results can be found in Table 1. Examination reveals that extremely small message sizes show a notable improvement over the sum of the bandwidths, which we believe is due to the pipelining of the requests on the two devices. This advantage remains in place for messages up to 32 or 40 bytes in size. Larger messages from 40 to 56 bytes tend to show a slight improvement until the bandwidth of the gigabit device overtakes Round-Robin at 64 bytes.

CPU Utilization: The figures for CPU utilization show Fast Ethernet taking approximately 20-30% of the local and remote CPU. Any gigabit algorithm other than round-robin (Figure 7) has a graph virtually identical to Figure 6, indicating that, while the additional overhead from more sophisticated algorithms

Msg. Size	R.R. Mbps	F.E. & G.E.	% Impr.
1	4.4	2.8	155.7
8	46.3	29.6	156.3
16	84.6	74.8	113.0
24	117.0	82.6	141.6
32	143.8	111.2	129.2
40	147.0	114.7	128.1
48	140.0	143.2	97.7
56	149.2	129.7	115.0
64	144.1	148.6	97.0
72	149.7	152.9	97.9
80	141.3	165.3	85.4

Table 1. Percentage improvement of Round-Robin above the sum of the individual Fast Ethernet and Gigabit Ethernet devices alone.

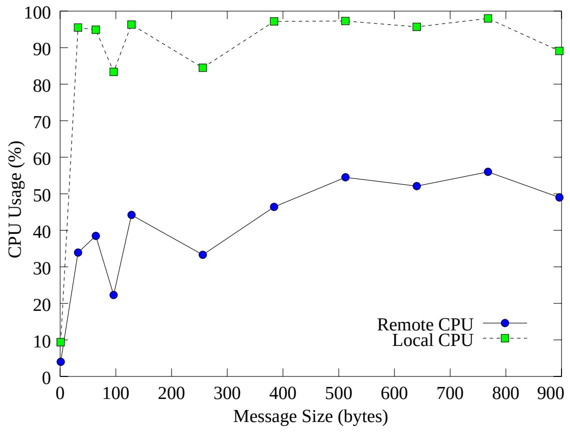

Figure 6. CPU utilization for Gigabit Ethernet.

may not be noticable, TCP/IP traffic at gigabit speeds is not going to leave much CPU available for other tasks.

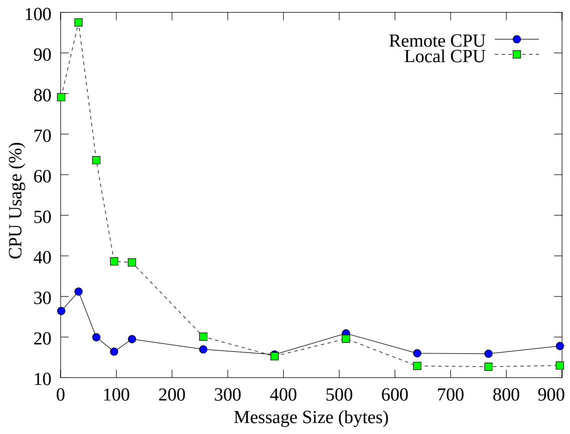

Figure 7. CPU utilization for Round-robin Ethernet.

5 Future work and conclusions

Efficient communication in distributed systems is essential for optimal system performance. As parallel computation develops, many different kinds of messages need to be sent between machines. These messages may have different requirements for network latency and bandwidth. In this paper, we revisit combining two physical interfaces into one virtual network (channel bonding).In contrast with most existing implementations, we communicate via *heterogenous* interfaces, and explore a number of more sophisticated scheduling algorithms than the round-robin technique used in the standard Beowulf channel-bonding driver. We explore the relative tradeoffs of interface scheduling overhead versus the additional bandwidth possible with multiple interfaces. Using both Gigabit Ethernet and Fast Ethernet Networks, we evaluate system performance using netPerf. We show significant performance gains for small messages, with incremental improvements for larger packets.

In the future we are looking forward to evaluating other combinations of networks, such as gigabit Ethernet and Myrinet under MPI (Message Passing Interface). We also hope to optimize our implementations of the algorithms above to reduce their CPU overhead and enhance their throughput.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under the award numbers CCR-0082667 and ACS-0092839. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also wish to thank Basong Zhao for his previous work on channel bonding.

References

[1] Daniel Andresen and Zhao Baosong. Heterogeneous channel bonding on a Beowulf cluster. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, pages 2479–2484, Las Vegas, June 2000.

[2] M. A. Blumrich, R. D. Albert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner. Design choices in the SHRIMP system: An empirical study. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*, 1998.

[3] Jenwei Hsieh, Tau Leng, Victor Mashayekhi, and Reza Rooholamini. Architectural and performance evaluation of giganet and myrinet interconnects on clusters of small-scale SMP servers. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, TX, November 2000. IEEE and ACM SIGARCH. Dell Computer Corp.

[4] JunSeong Kim and David J. Lilja. Utilizing heterogeneous networks in distributed parallel computing systems. In *Proceedings of the Sixth IEEE Intnl. Symp. High Performance Distributed Computing HPDC*, pages 336–, Portland, OR, August 1997.

[5] John Mehaffey. Highly available networking. *Embedded Linux Journal*, 7:45–47, January/February 2002.

[6] Bryan Pfaffenberger. *Linux networking clearly explained.* Academic Press, New York, NY, USA, 2001.

[7] Chance Reschke, Thomas Sterling, Daniel Ridge, Daniel Savarese, Donald J. Becker, and Phillip Merkey. A design study of alternative network topologies for the beowulf parallel workstation. In *Proceedings of the Fifth High Performance Distributed Computing (HPDC '96)*, pages 626–636, August 1996.

[8] R. W. Smith. *Advanced Linux Networking.* Addison_Wesley, June 2002.

[9] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, August 1995.