The Design and Implementation of ZCRP
Zero Copying Reliable Protocol

Mikkel Christiansen Jesper Langfeldt Hagen Brian Nielsen
Arne Skou Kristian Qvistgaard Skov

August 24, 1998

1 Design

1.1 Service specification

We begin by summarizing the services to be provided by the protocol and then describe the overall ideas of how these are to be provided.

- **Efficient transfer of datagrams over ATM network.** The purpose of the protocol is to transfer datagrams over ATM networks.
- **Simplex communication.** The protocol is only to provide the user with functionality for one way transfer of messages. If two way communication is needed, the user needs to establish a second connection in the opposite direction.
- **Reliable communication.** The service provided by the protocol is to be reliable. When a message is transferred and handed to the receiver, the protocol is to guarantee the integrity of the transferred data.
- **Flow and congestion control.** To adapt to the physical characteristics on which the protocol is to be used, the protocol is to include flow and congestion control.
- **Minimal copying.** To support efficient transfer of data the copying of data is to be minimized.
- **Block acknowledgements.** To ensure efficient reliable communication acknowledgements of received datagrams are to be sent in blocks.
- **Asynchronous and synchronous communication.** The protocol is to provided both block and non-blocking communication.

1.2 Achieving efficiency

The basic idea, which is to provide the protocol with efficiency and speed is based on block-acknowledge of received data blocks and minimal copying of data. ZCRP is designed to allow data transfer to continue without disturbance as blocks of positive or negative acknowledgements are transferred. This functionality is provided through the use of parallelism to handle outgoing and incoming data and by attaching information about next coming data in transfer units.

1.2.1 Achieving minimal copying

The amount of copying is kept to a minimum through the use of a zero-copying scheme [TK95]. Zero-copying is provided by using attached information in each data-block which is transferred. The basic idea of zero-copying is provided by having precise information about where the next coming data is to be placed. In this way a specific input buffer is usually not needed.

1.2.2 Achieving synchronous and asynchronous communication

This functionality is handled by buffering send and receive buffers and then letting the protocol handle data transfer when the necessary synchronization has been performed.

1.2.3 Achieving flow control

The protocol is to include a window mechanisms which can control the flow of datagrams to be transferred. If too many datagrams are missing positive acknowledges, the protocol is to slow down transmission accordingly. Similarly if acknowledges are returned regularly the protocol is to transmit datagrams without limitations.

1.2.4 Achieving reliability

The protocol is to ensure reliability by returning acknowledges of received datagrams and retransmitting units which has not been correctly received.

1.3 Assumptions about the environment

The protocol is to be executed in an environment consisting of two users (a sender and a receiver) and two processors connected with a FORE based ATM network as transmission channel. The sender request the sending of a message and the receiver request the reception of a message. These requests can be both blocking (synchronous) and nonblocking (asynchronous) in which case the user must be supplied with ways of synchronizing for the completion of the request.

It is assumed that the transmission channel looses, reorders and duplicates messages. It is also assumed that any distortion of messages is handled by the underlying ATM protocol and that no spontaneous insertion of irrelevant messages is done. The FORE API which provides access to the ATM protocol is used and the protocol is to be based on Adaptation Layer 5 of the ATM protocol. This motivates the correctness of our assumptions.

The ATM network consists of the following elements:

- Two FORE ATM cards model SBA-220.
- Two FORE ATM cards model SBA-220e.
- One FORE ATM switch model ASX-200 with four ports.

The network is connected by unshielded twisted pair cable conforming to the UTP category 5 standard. Every card and switch port has a network bandwidth of 155 Mbps.

Processors used by the protocol can have varying speeds. In our setup we have the Sun microsystems workstations shown in table 1 to our disposal. As shown in the table we have processors varying in size from very small to very large and we have both single and dual processor systems. An important goal of the protocol is to efficiently utilize any of these setups.

Name(s)	*Model*	*Processor(s)*	*Memory size*	*ATM card*
Beta, Spring	SPARCstation 5	70 MHz microSPARC II	48 Mb	SBA-220e
Ahorn, Birk	SPARCstation 20	2 × 50 MHz SuperSPARC	96 Mb	SBA-220
Altair, Sirius	Ultra 1	143 MHz UltraSPARC	64 Mb	SBA-220e
Spock	Ultra 2	2 × 168 MHz UltraSPARC	256 Mb	SBA-220e
Kirk	Ultra 2	2 × 168 MHz UltraSPARC	512 Mb	SBA-220e

Table 1: The various workstations used in out setup.

This will ensure that the protocol can be broadly used on various platforms regardless of size and speed.

1.4 ZCRP API

The basic primitives of communication in the ZCRP protocol is provided by the ZCRP API. This is shown in table 2. The API provides functions for message allocation, protocol initialization and synchronous and asynchronous communication. These will be explained thoroughly in the following.

Function	Sender	Receiver
Initialize protocol	zcrp_send_init	zcrp_rcv_init
Allocate message	zcrp_alloc_msg	zcrp_alloc_msg
Free message	zcrp_free_msg	zcrp_free_msg
Communicate message	zcrp_send	zcrp_rcv
Wait for communication termination	zcrp_wait	zcrp_wait
Probe for communication termination	zcrp_probe	zcrp_probe

Table 2: The ZCRP API.

The data structure representing a message is the `zcrp_msg` structure:

```
struct zcrp_msg {
  nbyte *buffer;
  ...
  nlong message_size;
  nlong buffer_size;
  ...
};
```

The `buffer` is the area holding the actual message. The `message_size` represents the size of the message, and the `buffer_size` represents the size of the buffer, which is larger than the message size because of the extra space needed for trailer information.

Messages are allocated and freed with the allocation primitives:

```
struct zcrp_msg *zcrp_alloc_msg (nlong size);
void zcrp_free_msg (struct zcrp_msg *message);
```

The `zcrp_alloc_msg` allocates a message of size `size` and initializes the above mentioned fields in the message structure. A message can be used for sending and receiving any message as long

as the message can be fitted within the buffer (the size is smaller than or equal to the message size.) The primitive returns `NULL` on failure. Messages are freed with the `zcrp_free_msg` primitive.

In order to start up the protocol entities on either side, the initialization primitives are provided:

```
nbyte zcrp_send_init (int *file_descriptors);
nbyte zcrp_rcv_init (int *file_descriptors);
```

The `file_descriptors` points to a 2-element array of file descriptors. The first is the descriptor to use for the communication channel from receiver to sender (carries block acknowledgement and request to send datagrams) and the second descriptor represents the communication channel from sender to receiver (carries units.) These can be simplex channels. A single duplex channel can also be used, in which case the same file descriptor is provided in both positions of the array. Both of these return 1 on success and 0 on failure.

Initiation of sending and receiving messages is done through the use of the communication primitives:

```
nbyte zcrp_send (struct zcrp_msg *message, nlong size);
nbyte zcrp_rcv (struct zcrp_msg *message, nlong size);
```

In both of these, a message is provided (`message`) and the size of the message to send/receive (`size`.) Both of these are nonblocking and returns immediately 1 on succes or 0 on failure.

Synchronization between sender and receiver is provided through the synchronization primitives:

```
nlong zcrp_wait (struct zcrp_msg *message);
nbyte zcrp_probe (struct zcrp_msg *message);
```

The `zcrp_wait` blocks until the specified message is sent or received (depends on who makes the call.) It returns 1 on success and 0 on failure. The `zcrp_probe` is the nonblocking alternative which simply returns a boolean indicating whether or not the transmission has ended.

An example use of these primitives is shown in figure 1. The figure also shows an example of the traffic generated by the protocol.

1.5 Protocol vocabulary

Internally ZCRP includes three types of messages. These are:

- *message* — which is a total block of data to be transferred.
- *acknowledgement* — which is used for returning block acknowledgements which includes information about the state of received data.
- *request to send* — which are used for performing the necessary synchronization between the sender and receiver protocol entities before data is actually transferred.

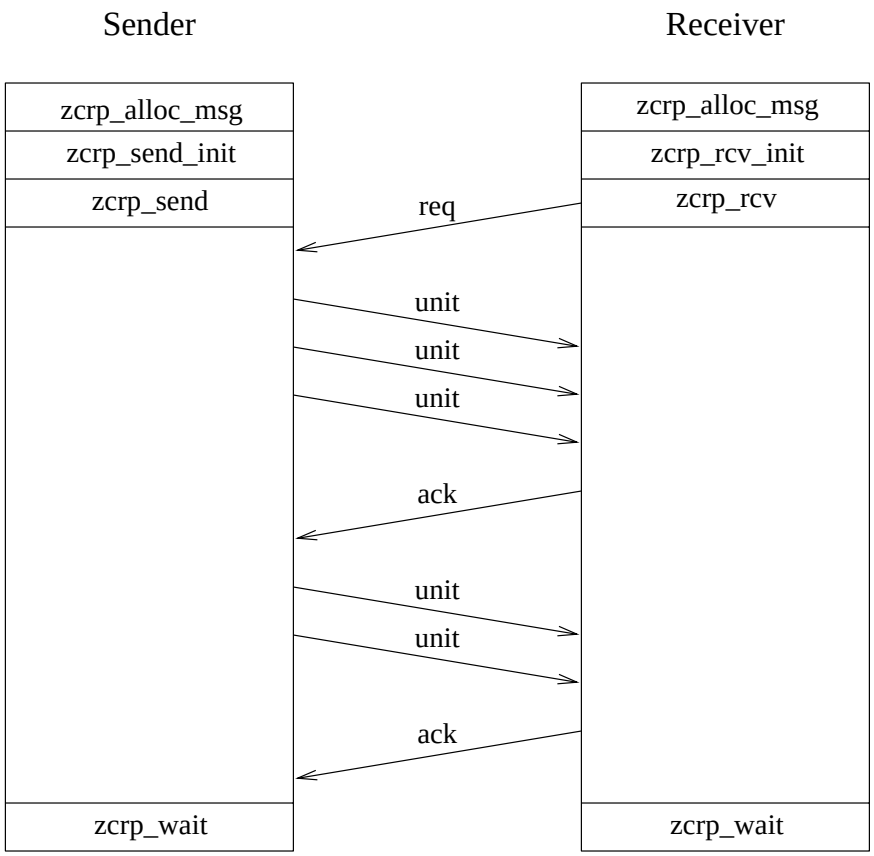

Figure 1: An example of the use of the ZCRP API. The traffic generated by the protocol is indicated by arrows.

1.5.1 Message and message number

At the highest level of abstraction is the *message*. This is the only entity of ZCRP that shines through to the user of the protocol. It is simply a block of data with two attributes: *identity* and *size*. The identity of a message is also called the *message number* or *MN* for short. The identity is handed to the user and can be used to check the status of a message being transferred.

1.5.2 Unit and unit number (PDU)

Since the FORE API can only send datagrams of sizes up to MTU, the protocol has to perform fragmentation of *messages* into *unit datagrams* of size MTU. The unit includes a datafragment to be transferred and information identifying the unit. Units are not fragmented further by ZCRP and represents actual protocol datagram units (PDU). This is explained in detail in section 1.6.

1.5.3 Block acknowledgement (PDU)

Positive and negative acknowledgements are buffered in the receiver protocol entity (PE) and returned to the sender protocol entity in blocks. The *acknowledgment* datagram is used for this. By returning acknowledgements in blocks the overhead necessary to provide reliable datatransfer is kept to a minimum.

1.5.4 Request to send (PDU)

Before data can be actually transferred both the sender PE and receiver PE need to have the necessary send and receive buffers ready. The necessary synchronization is handled by the *request to send* datagram which is sent by the receive PE indicating the receiver is ready to receive a set of identified messages.

1.6 Datagram formats

1.6.1 Unit

The unit datagram represents a fragment of a message. As mentioned in section 1.5, every message is fragmented into unit datagrams of size MTU, since the ATM layer can only send datagrams of size less than or equal to MTU.

In figure 2 the unit datagram is shown. The *data* corresponds to the fragment of data from the message. In order to identify this data, each unit has a *message number* and a *unit number*. The message number identifies the message from which the unit belongs and the unit number indicates the number of the unit within the message. The *length* is the length of the unit. In every unit except the last in a message, this will be the same, namely the fragment size. In the last unit of a message, this can be smaller than the fragment size, in which case the *padding* is filled with meaningless data until the unit datagram has a size of MTU.

In order to handle positive and negative acknowledgement, each unit is assigned a unique *sequence number*. Sequence numbers are taken from the increasing sequence of natural numbers. When a unit is received, the sequence number attached can be sent back to the sender in a positive acknowledgement. The sender can then determine exactly which unit has been received. Negative acknowledgements can easily be calculated. If some sequence numbers are missing between two successively received sequence numbers, negative acknowledgements can be sent for these. The sender can then retransmit the units in question.

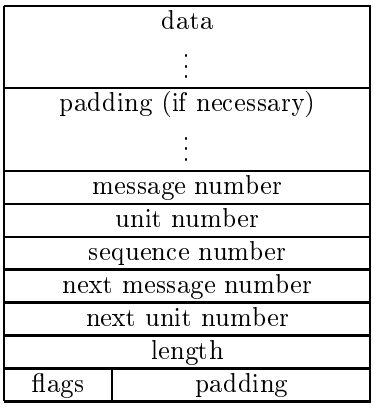

Figure 2: Unit datagram.

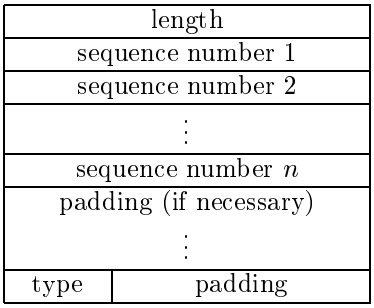

Figure 3: Block acknowledge datagram.

Zero-copying is achieved by always attaching a unit with the identity of the next unit to be sent. This information is stored in the *next message number* and *next unit number* attributes. Having this information, the receiving side can prepare the buffer space in which to receive the next unit. This way, the unit is received in place in most situations.

The *flags* field in a unit has two flags: *has next* and *is last*. The first is set if the *next message number* and *next unit number* has been set. In some cases, there is no information on the next unit to be sent. The latter is set if the unit is the last in a message. This is how the sender tells the receiver the size of the message. Alternatively we could have attached the message size to all units, but this would have added an unnecessary overhead to each unit.

1.6.2 Block acknowledge

Instead of sending positive and negative acknowledgements separately, several are combined into a block acknowledge datagram. This is seen in figure 3.

A series of acknowledgments of equal sign (positive or negative) can always be combined into an interval $[a, b[$ with a being the first sequence number and b being the sequence number right after the last sequence number. The block acknowledge datagram consists of a sequence of intervals of alternating sign and the number of intervals is indicated by the *length* attribute. The reason for alternating the sign is the simple fact that any transmission will ultimately be an alternation between sending and dropping datagrams. In the figure is shown the representation of the intervals

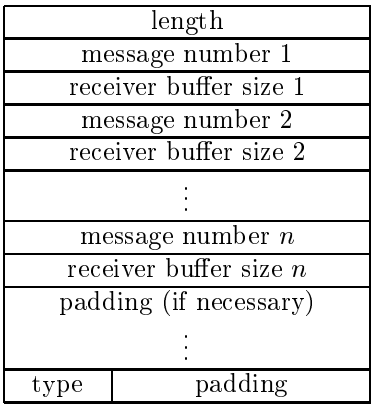

Figure 4: Request to send datagram.

- [sequence number 1, sequence number 2[,
- [sequence number 2, sequence number 3[,
- ..., and
- [sequence number $n-1$, sequence number n[.

The sign of the first interval is indicated by *type* field in the datagram which has one of the values *positive* and *negative*.

1.6.3 Request to send

The protocol is receiver initiated. This means that a message is not sent until the receiver is ready. Zero-copying is the reason for this. If the receiver is ready when the sender initiates the transmission, no buffer is needed on the receiver side. The message can be received in place.

The way the receiver informs the sender of readiness is by the use of the *request to send* datagram. It is shown in figure 4.

The datagram consists of a sequence of pairs of message number and receiver buffer size. The length of this sequence is indicated by the *length* attribute. The *type* attribute has the value *request*.

Since the block acknowledgement and request to send datagrams are only to be sent from the receiver to the sender, they are defined to have the same length and share communication channel. The value of the *type* field indicates whether a datagram received on the sender side is a block acknowledgement (by having values *positive* or *negative*) or a request to send (by having value *request*.)

1.7 Procedure rules

Based on the previous descriptions we will now describe the algorithms responsible for the internal flow of messages and units. The following section does not go into many of the specific details — we just give an overview of the protocol. We begin by giving an informal description of the the sender protocol entity (PE), the receiver PE and then continue with a description of state diagrams for the threads which are used in ZCRP.

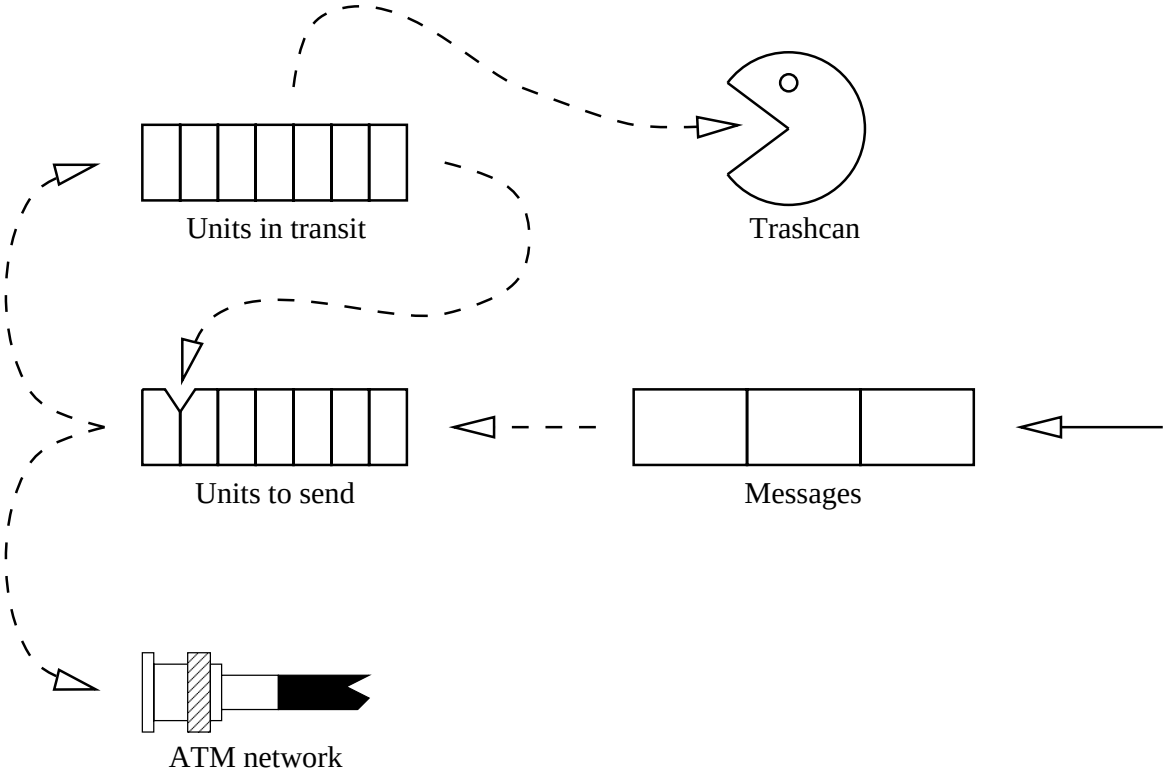

Figure 5: An abstract view of the Sender Protocol Entity.

1.7.1 Sender PE

Figure 5 shows an abstract view of the sender PE. ZCRP allows non-blocking communication and due to this functionality the protocol needs mechanisms for queueing messages.

The user can issue several sends without worrying about the synchronisation with the receiver PE. This is handled by the protocol. When the user calls zcrp_send, the message is automaticly stored and then used when the receiver PE is ready for receiving.

As described in section 1.5, the messages can only be transmitted through the FORE API in blocks of MTU size (units). Before a unit is sent it is tailored with the necessary information needed for identification.

Upon sending a unit it is transferred to a list of *units in transit*. The sender PE of ZCRP waits for acknowledgements for each unit and if a unit has not been received properly it is retransmitted. This is done by removing the unit from *units in transit* to *units to send* as illustrated by the figure. For reasons of efficiency retransmissions have high priority and are placed first in *units to send*.

Acknowledgements from the receiver PE are received in blocks, but ZCRP does not stop and wait for acknowledgements to arrive. If there are messages ready to be sent and the receiver is ready to receive, the sender PE continually sends units. ZCRP includes a sliding window mechanism if that too many units are waiting for acknowledgement.

The transfer of Messages and Units is only done on a conceptual basis; zero copying. No memory is copied.

1.7.2 Receiver PE

The ZCRP user interface both facilitates non-blocking send, zcrp_send and non-blocking receive, *zcrp_recv*. Synchronization is handled by the protocol. When calling zcrp_recv a buffer is handed to the receiver PE and then ZCRP transmits synchronization information about the specific message to the sender PE.

The constant flow of units to the receiver PE speeds up the transmission of messages by reducing the overhead needed for making the protocol reliable. However in order to handle the steady stream of units the receiver PE needs an intelligent algorithm for receiving units. This algorithm is based on a principle of *predicting* which units are received and is then used together with a zero-copying mechanism.

As described in section 1.6 each unit includes a trailer with information, not only about the specific unit, but also about the next unit expected. With this information available the receiver tries to guess which receive buffer to use and then fills in units from the sender PE. If a received unit turns out to be out of sequence it is copied to the right buffer. This scheme of course has special cases such as first and last units of a message and units received out of order. But this is all handled by the ZCRP.

The processing of received units is handled by a single thread. This ensures that trailers are not overwritten by a new received unit before the information in the trailers has been processed.

Based on the trailer information from the received units the receiver PE regularly returns information about successful and unsuccessful transmissions. As mentioned a sequence number is used for each unit to keep track of the transmitted units. A timeout mechanism is included on both receiver PE and sender PE for handling the case of lost units. When a message has been successfully received by the receiver PE information about this is also transmitted to sender PE.

1.7.3 Threads and ADT

Having given an informal description of the sender PE and receiver PE we now turn a detailed description of the use of threads in ZCRP and their functionality. ZCRP uses a total of 7 threads organized in 4 threads in the sender PE and 3 in the receiver PE. In figures 6 to 12 the state diagrams for each thread is given. These diagrams describe the main functionality but do not include all details of the protocol.

User thread — sender PE The user thread on the sender PE is responsible for user interaction and initiating data transfers after proper synchronization. A state diagram is seen in figure 6. When a request is received for sending a message, the user thread checks for synchronization, as described in 1.5 a message is fragmented and handed to the sending thread. If a zcrp_wait call for a message is received the thread blocks until the message is received.

Sending thread – sender PE The sending thread has a simple functionality in ZCRP. This thread is responsible for sending the actual units to the network and providing these with the necessary trailer information. This is illustrated on figure 7.

Update thread – sender PE This thread (see figure 8) handles incoming requests to send, positive and negative acknowledgements. When positive acknowledgements are received the status of the corresponding unit is updated. If negative acknowledgements are received, update thread initiates a retransmission of the units. Negative acknowledgements are generated if units are received out of sequence. A request for synchronization is handed to the user thread.

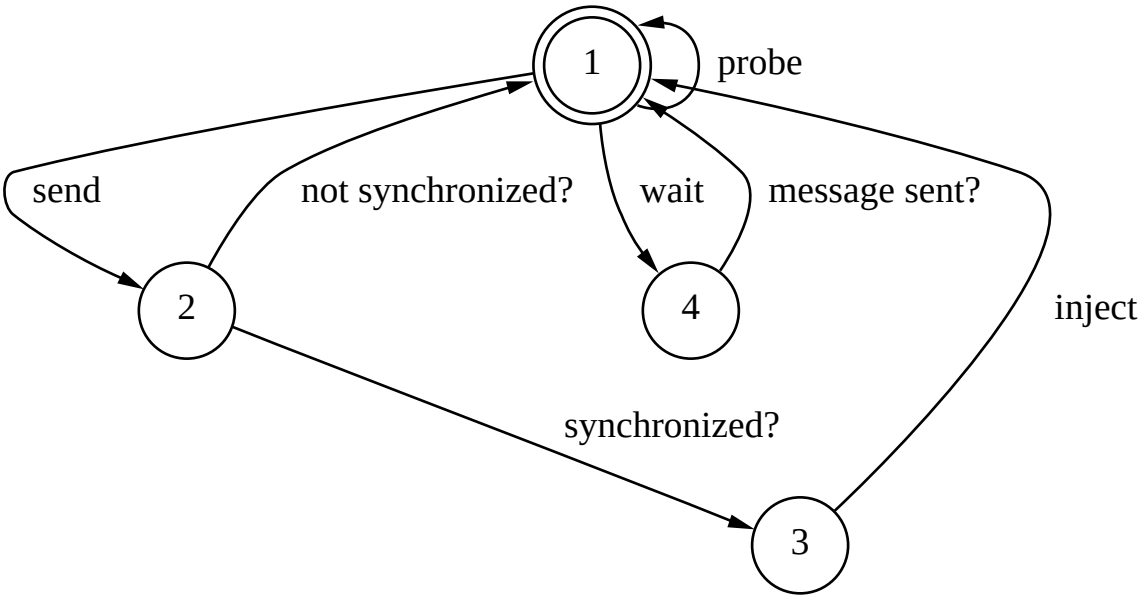

Figure 6: User thread — sender PE.

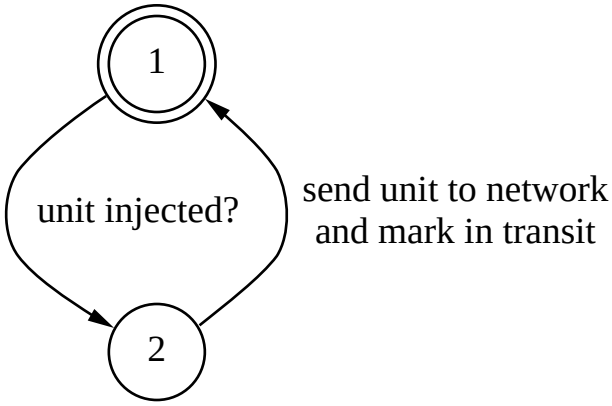

Figure 7: Sending thread — sender PE.

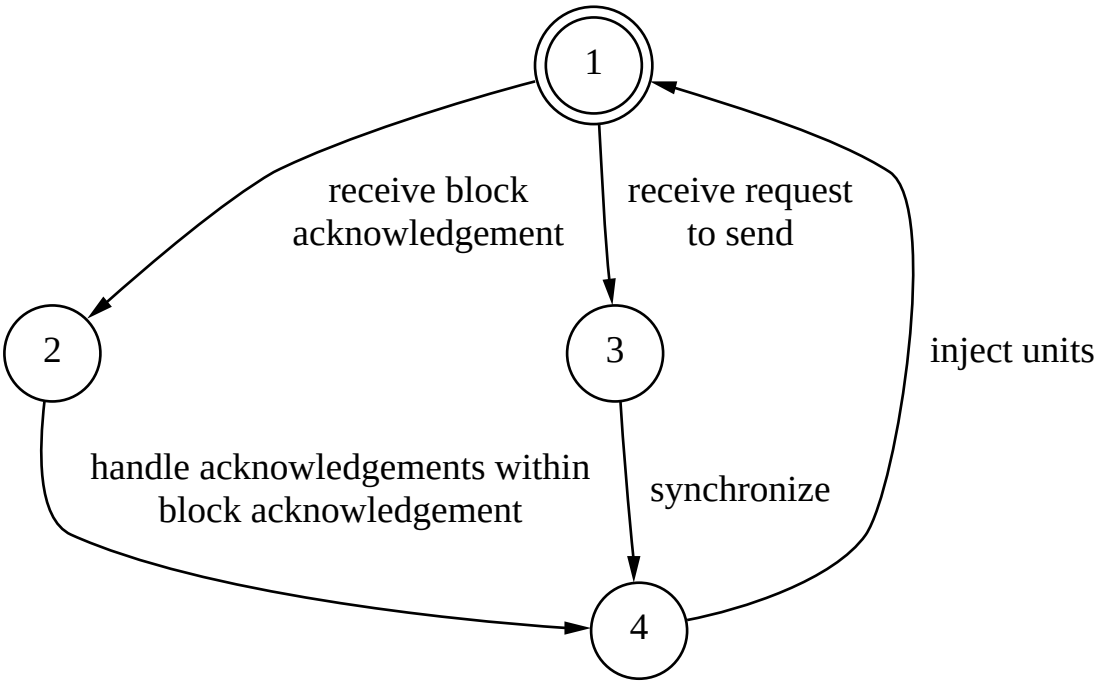

Figure 8: Update thread — sender PE.

Timeout thread – sender PE If a unit has been waiting too long for acknowledgement this thread is responsible for initiating a retransmission.

The threads uses two central ADT's representing *units to send* and *units in transit* on figure 9. The ADT representing *units to send* contains units ready to be sent to the network and the ADT for *units in transit* keeps information about each unit that has been sent but not acknowledged.

ZCRP also uses ADTs for handling the administration of messages. Internally information is kept about each message reflecting whether it is just buffered to transmission, is being sent or has been fully acknowledged.

User thread – receiver PE Has similar functionality as the user thread in the sender PE. When a request for receiving a message is initiated by the user this thread sends a request to the sender PE. Similarly the user can issue a wait command which blocks until a message has been successfully received (figure 10).

Receiving thread – receiver PE The receiving thread handles all receiving of units and keeps track of the number of received units for each message. When a message has been fully received a corresponding semaphore is signaled. For each message a semaphore is used on both receiver PE and sender PE to indicate whether a message has been received or fully acknowledged. This is used with the primitive; zrcp_wait (figure 11).

Timeout thread – receiver PE This thread seen on figure 12 is responsible for sending reqeusts to send to the sender PE when the user has called zcrp_recv and for regularly sending update information of received units if this has not been done automaticly.

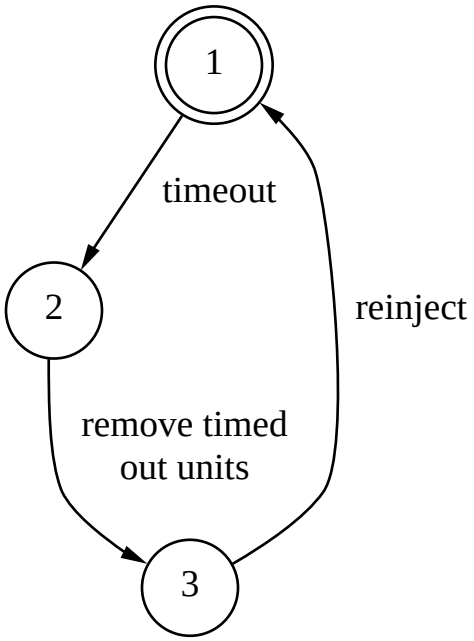

Figure 9: Timeout thread — sender PE.

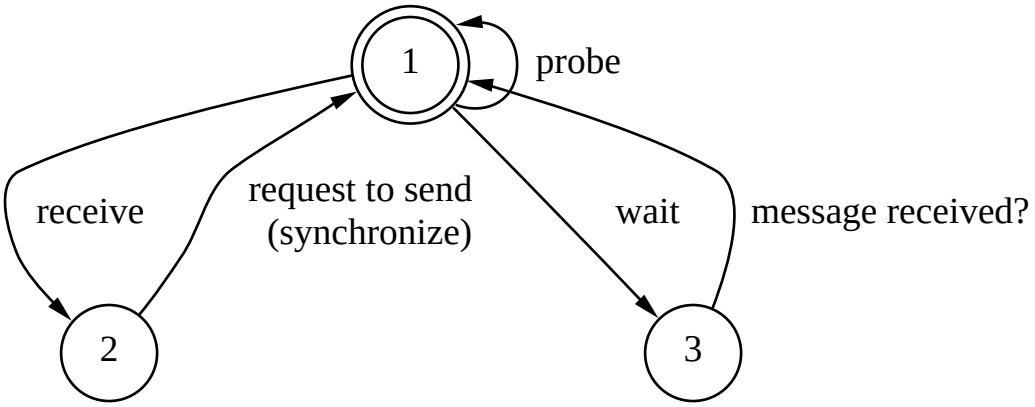

Figure 10: User thread — receiver PE.

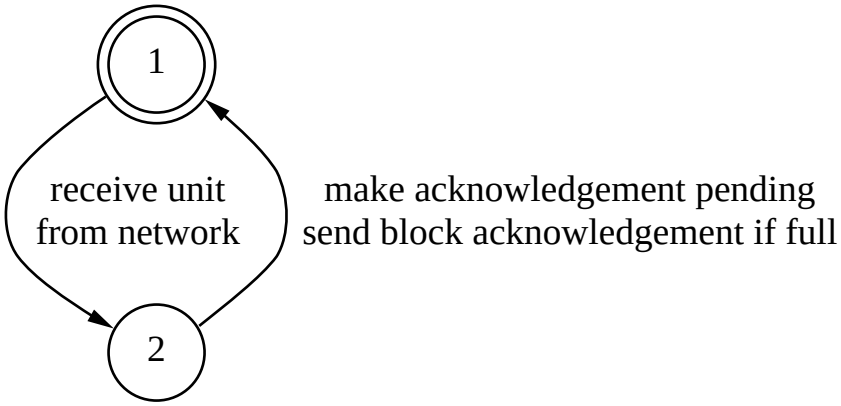

Figure 11: Receiving thread — receiver PE.

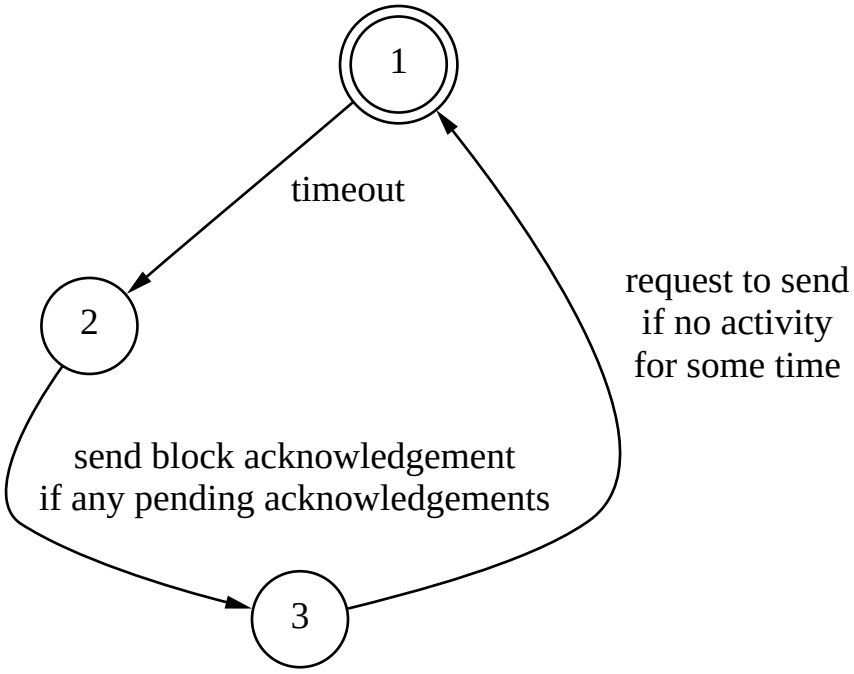

Figure 12: Timeout thread — receiver PE.

The threads in the receiver PE uses ADTs of the same type for handling units and messages.

2 Tests

In this section we describe the different tests or experiments that have been performed on ZCRP and TCP. The main goal of the tests has been to measure the efficiency of the protocols compared to network bandwidth and CPU resources.

First the test for measuring protocol throughput is described, followed by a message consistency test and finally we describe the test for measuring CPU usage.

A full test is built from a number of basic tests with different message size. In order to find an average performance, each basic test is repeated a number of times, typically five. It should be noted that the connection between the server and client is not closed during a full test.

To perform a basic test a message size and a number of messages is used. When the test is started, both the server and the client allocates memory for the number of messages that are needed for a measurement. The master calls zcrp_send for each message one message at a time. The client initiates the test by calling zcrp_rcv for each message, and then calls zcrp_wait for the last message. The watch is started just before calling zcrp_rcv the first time and stopped when zcrp_wait return, this means that the measurement is taken over a number of asynchronously sent messages of the same size.

In the case of TCP the full test is the same. The basic test uses write to initiate the sending of all messages, and the use read to receive the messages. The watch is started before the first write and after the last read. It should be noted that read can be called more times that the number of messages, because TCP does not guarantee that the whole message is delivered in one call.

In order to ensure that all messages are fully consistent after transmission, tests were made that checked the consistency of each message received. It was found that all messages were fully consistent. Since the test takes CPU cycles it was disabled during the other tests.

The ZCRP is designed to minimize CPU usage. By testing how well the protocol will perform on at loaded CPU, it is possible to tell whether this is true or not. To perform this test we designed a program that will do nothing but incrementing a counter, when the program receives a signal the value of the counter is written on the screen. The test was performed both on ZCRP and TCP in order to compare the amount of resources needed by the protocols. The results of this test is somewhat obscure. In the case of ZCRP it looks fine, because the protocol performance is almost unchanged. In the case of TCP the situation is quite strange, since the protocol performs better when the competitive process is running. Our opinion on this, is that it must be due to the Solaris scheduler.

2.1 The performance tests

1. In most cases ZCRP exceeds TCP/IP in performance when the message size exceeds some threshold (in the area of 50,000 bytes.) In the case of Ultra-II to Ultra-II, the TCP/IP protocol outperforms ZCRP with 20 Mbits/second.

2. In the cases involving Ultra and 5, we see equal behavior in ZCRP despite the difference in computing power present in the two platforms.

3. We see equal TCP/IP performance in Ultra to Ultra, Ultra to 5 and 5 to Ultra. It is therefore quite disturbing to see a smaller performance in the 5 to 5 test (roughly 20 Mbits/second smaller.)

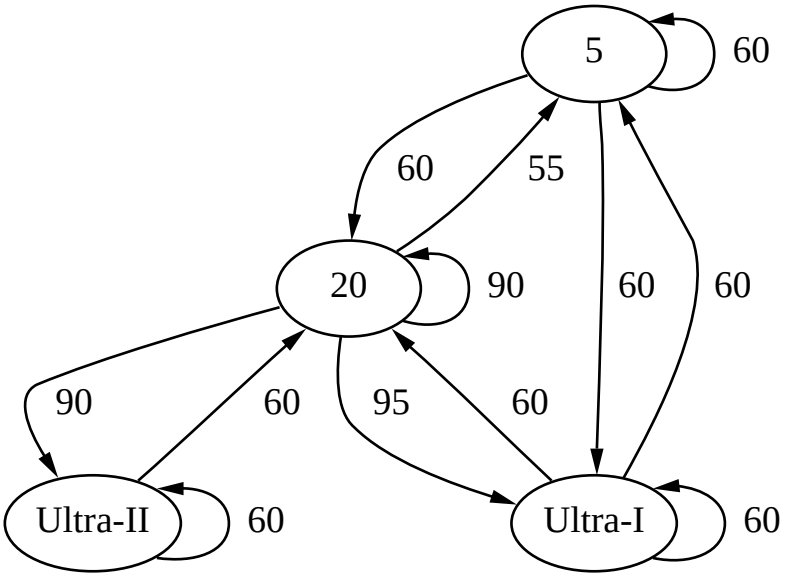

Figure 13: The approximate results of the ZCRP tests.

4. In the dual-processor configurations we see that both protocols are faster from 20 to Ultra-II than from Ultra-II to 20. In the ZCRP case we can conclude that the 20 outperforms the Ultra-II in sending with roughly 30 Mbits/second, while performance in reception is the same. We see some quite disturbing behavior in the TCP/IP results. From the Ultra-II to 20 and 20 to Ultra-II tests, we can conclude that both architectures can receive and send at a speed of roughly 60 Mbits/second. This contrasts with the results from the 20 to 20 tests, where we see that the speed is no higher than rougly 30 Mbits/second.

5. In the 5 to 20 and 20 to 5 we get rougly the same results. Sending is roughly 60 Mbits/second and reception is roughly 45 Mbits/second.

6. TCP/IP performance in 20 to Ultra and Ultra to 20 is lying at 40 Mbits/second. We see that the speeds from 20 to Ultra exceeds those from the Ultra to 20 test with roughly 30 Mbits/second.

The results are summarized in figures 13 and 14. Figures 15 – 30 illustrates the tests in more detail.

References

[TK95] Moti N. Thadani and Yousef A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical report, Sun Microsystems Laboratories, Inc., May 1995.

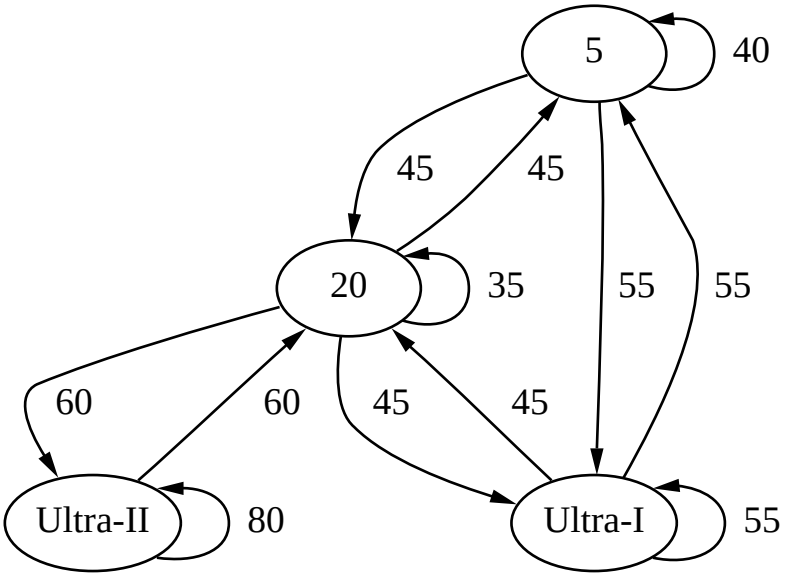

Figure 14: The approximate results of the TCP/IP tests.

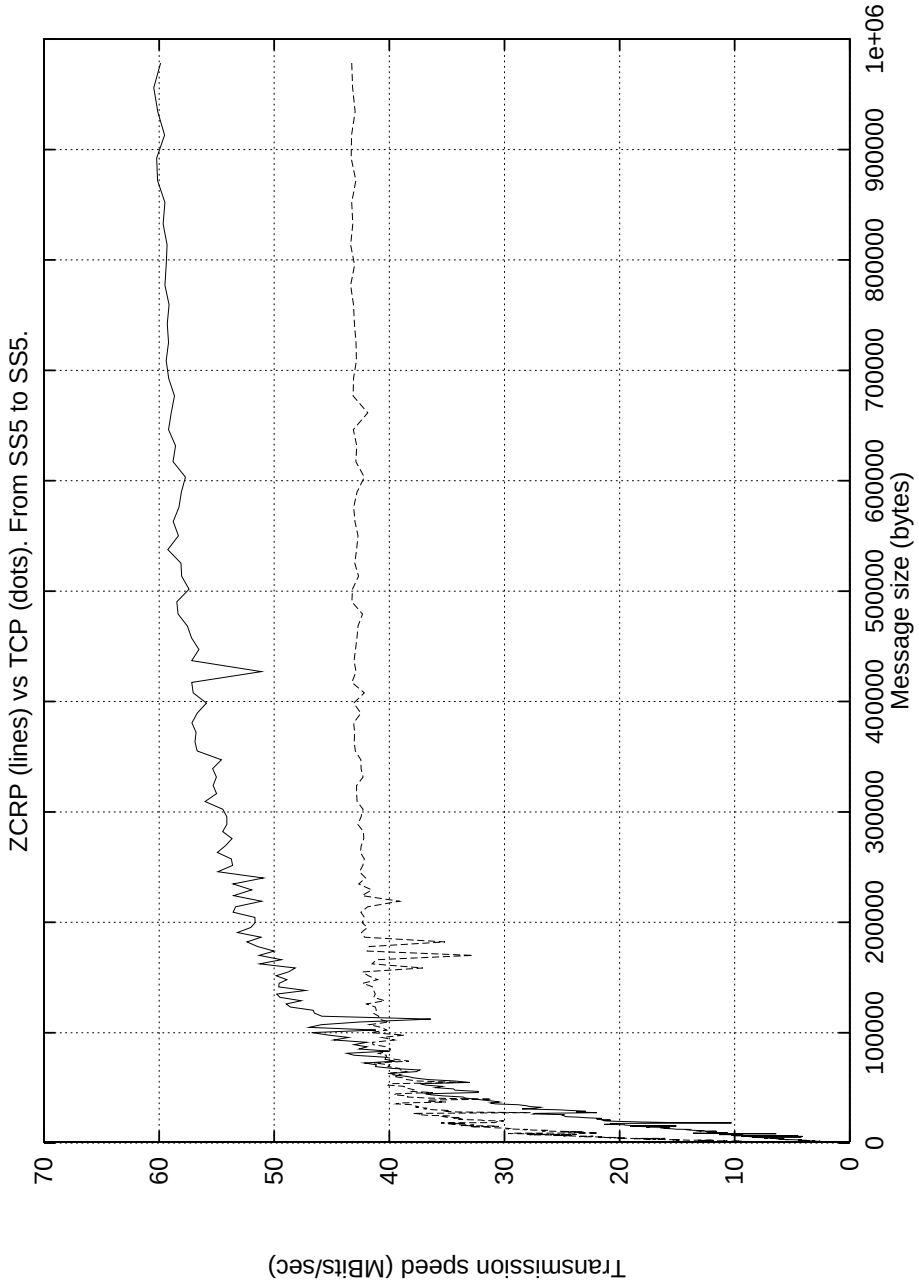

Figure 15: From Sparc5 to Sparc5.

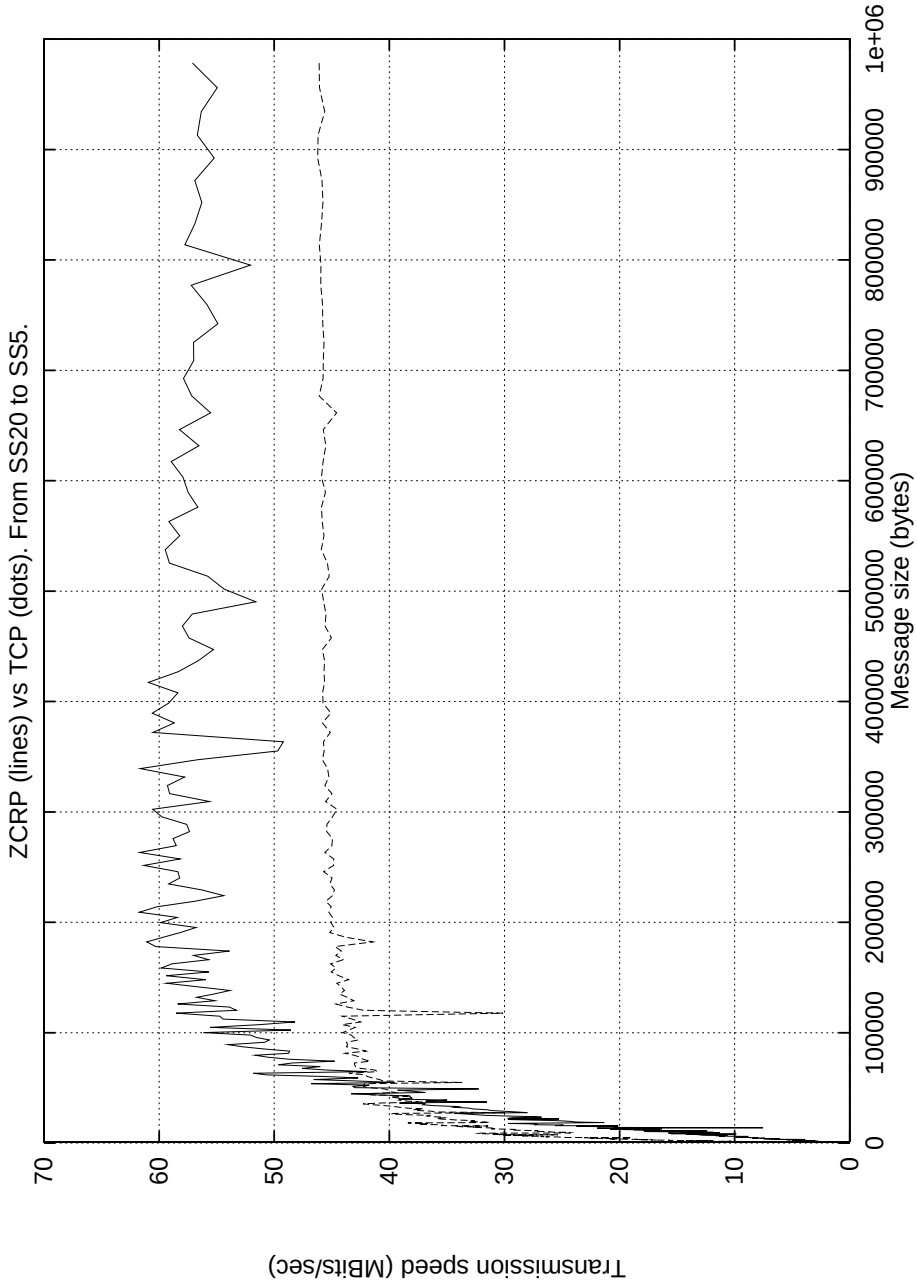

Figure 16: From Sparc20 to Sparc5.

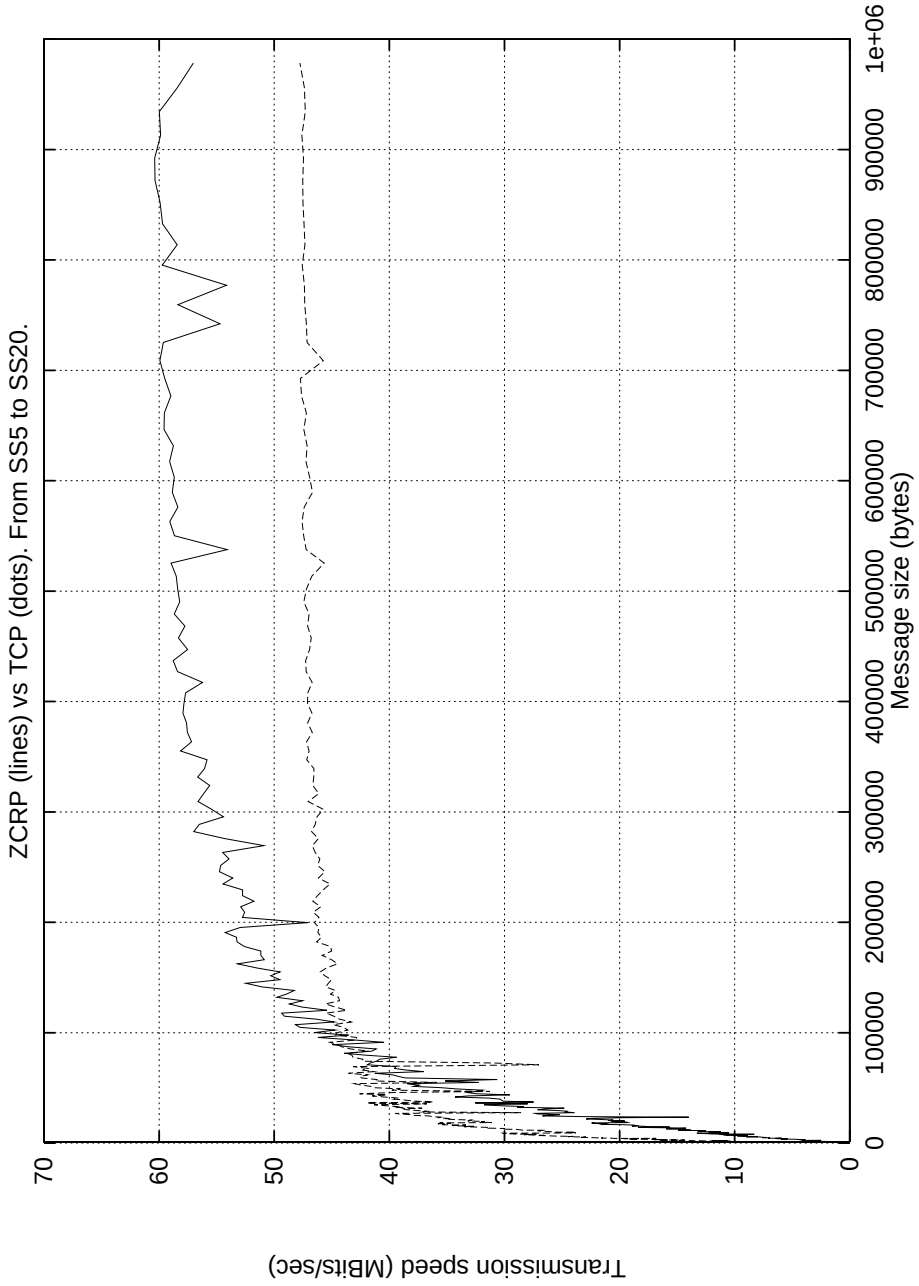

Figure 17: From Sparc5 to Sparc20.

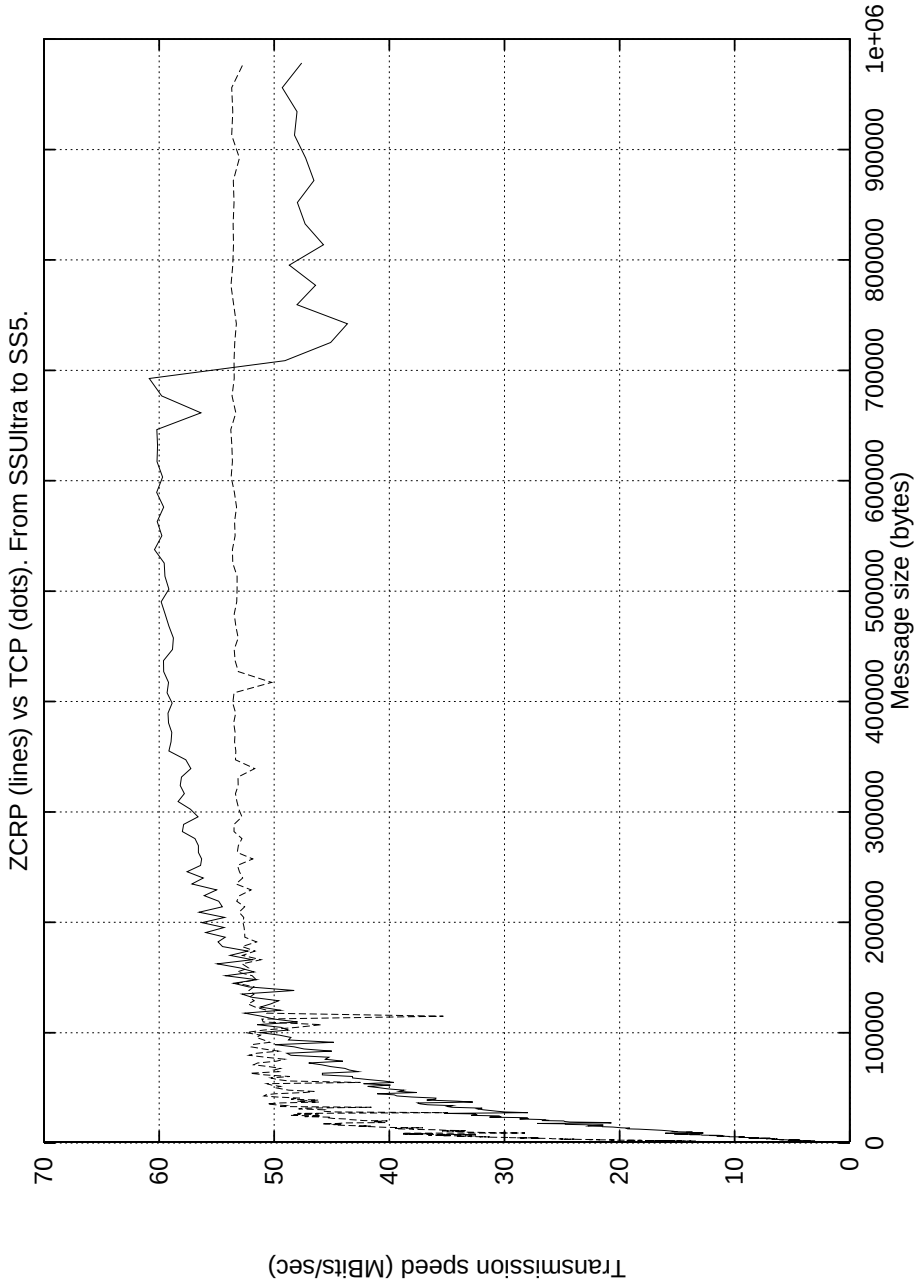

Figure 18: From SparcUltra to Sparc5.

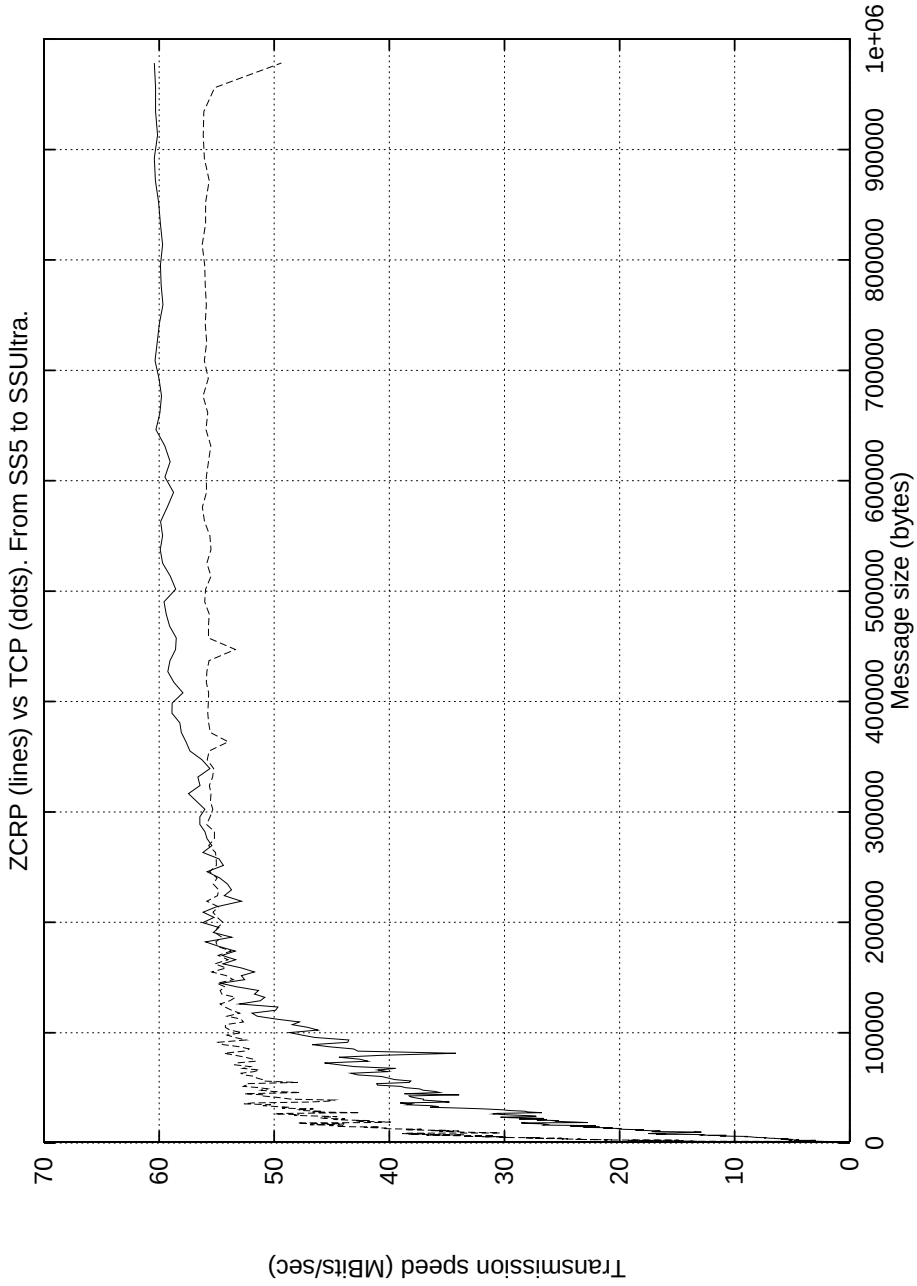

Figure 19: From Sparc5 to SparcUltra.

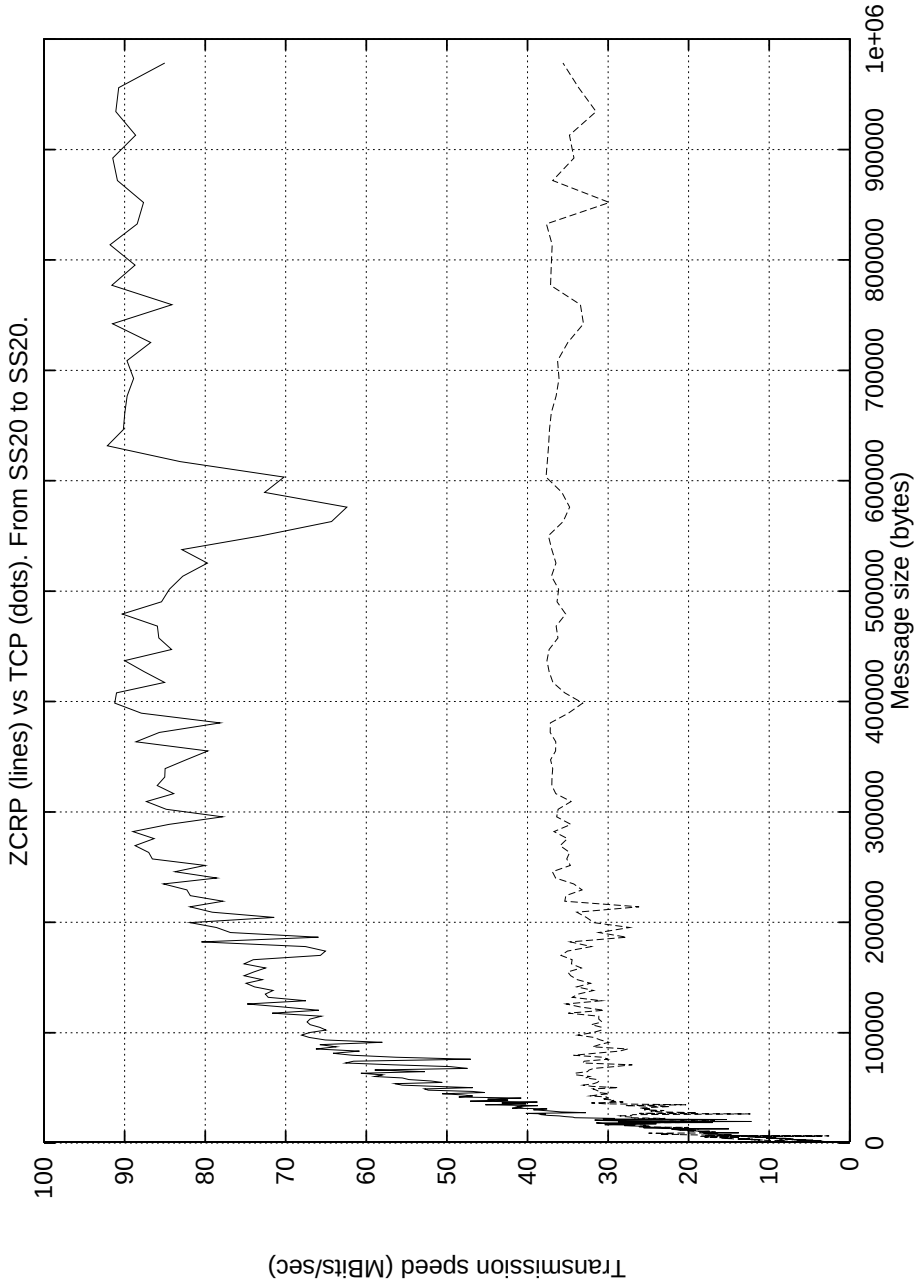

Figure 20: From Sparc20 to Sparc20.

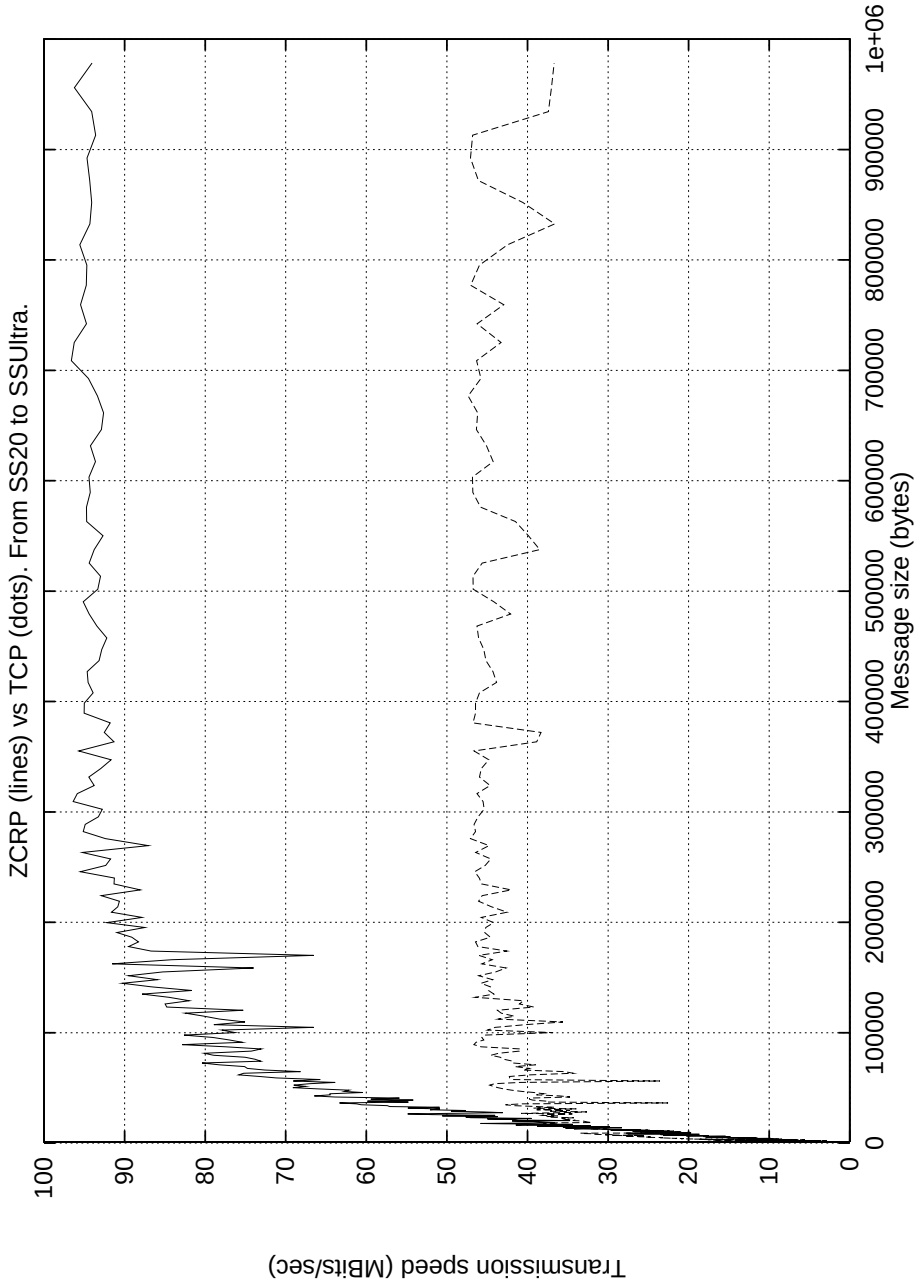

Figure 21: From Sparc20 to SparcUltra.

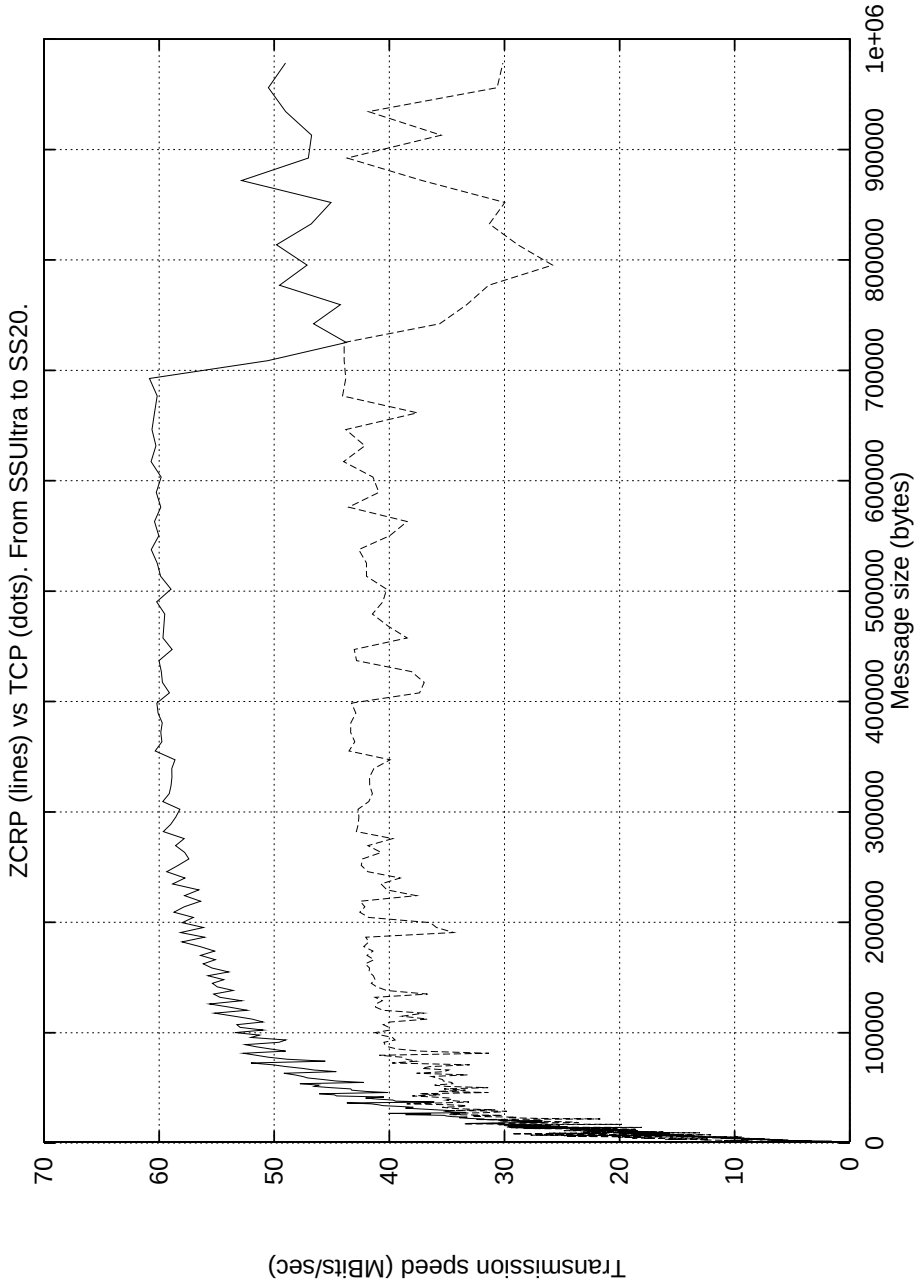

Figure 22: From SparcUltra to Sparc20.

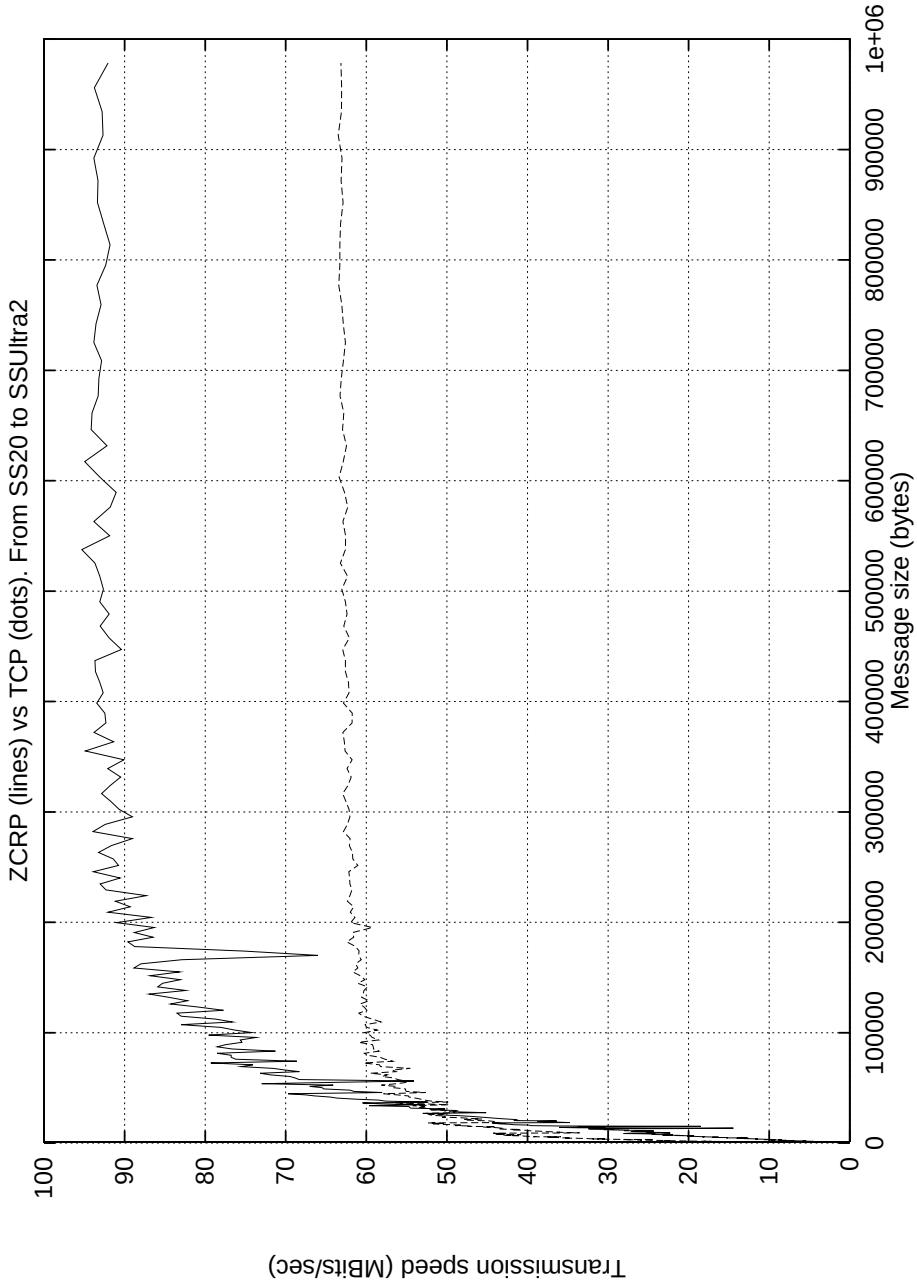

Figure 23: From Sparc20 to SparcUltra2.

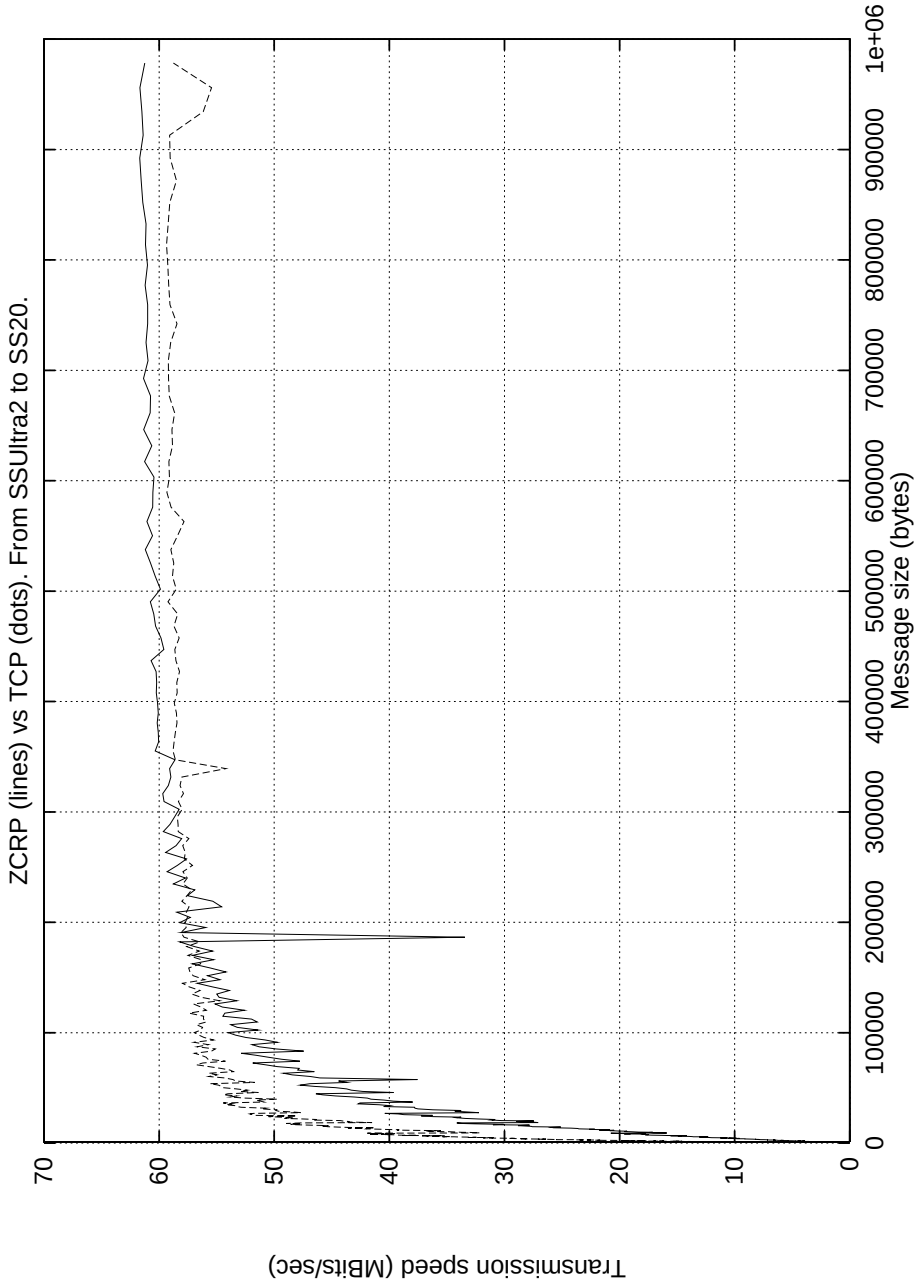

Figure 24: From SparcUltra2 to Sparc20.

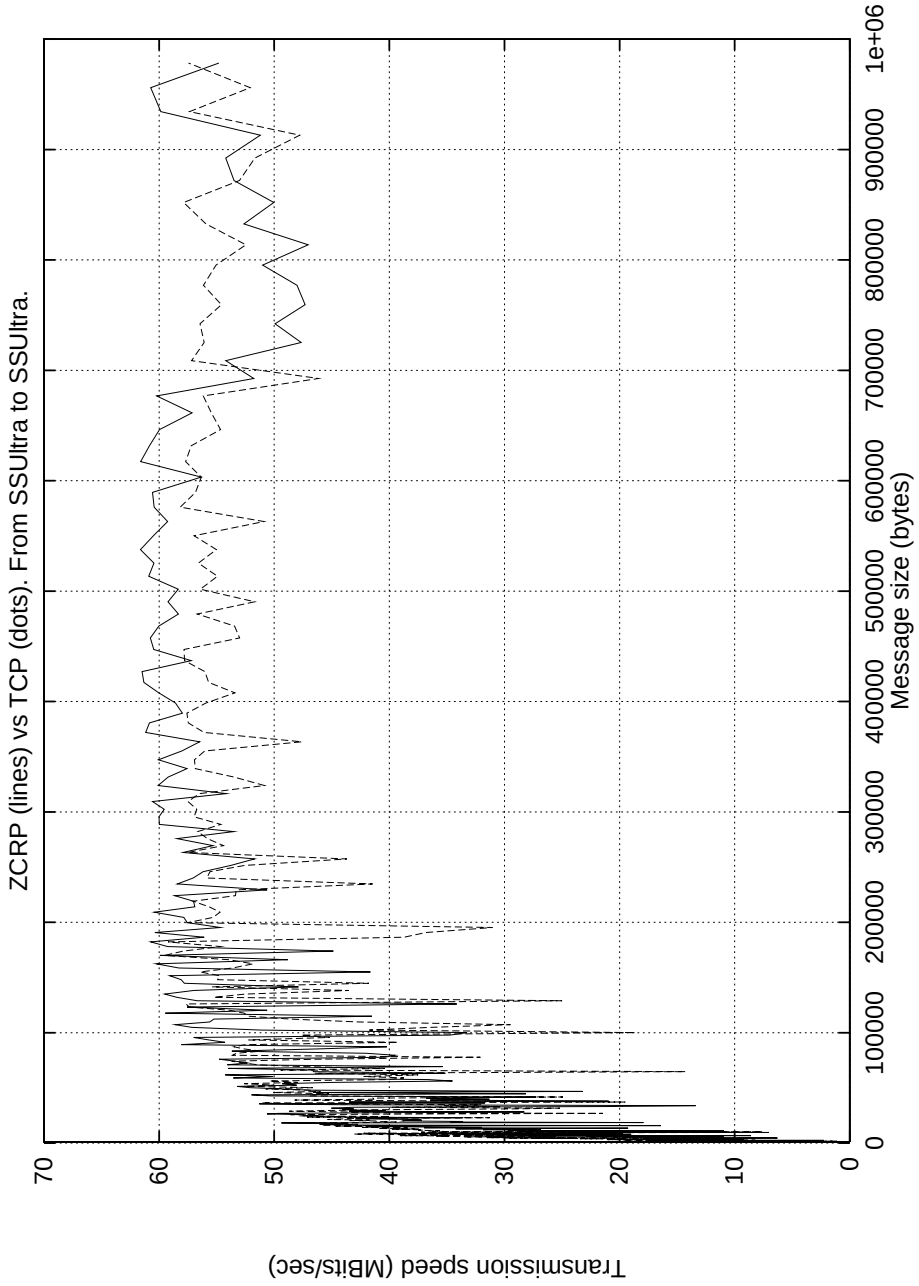

Figure 25: From SparcUltra to SparcUltra.

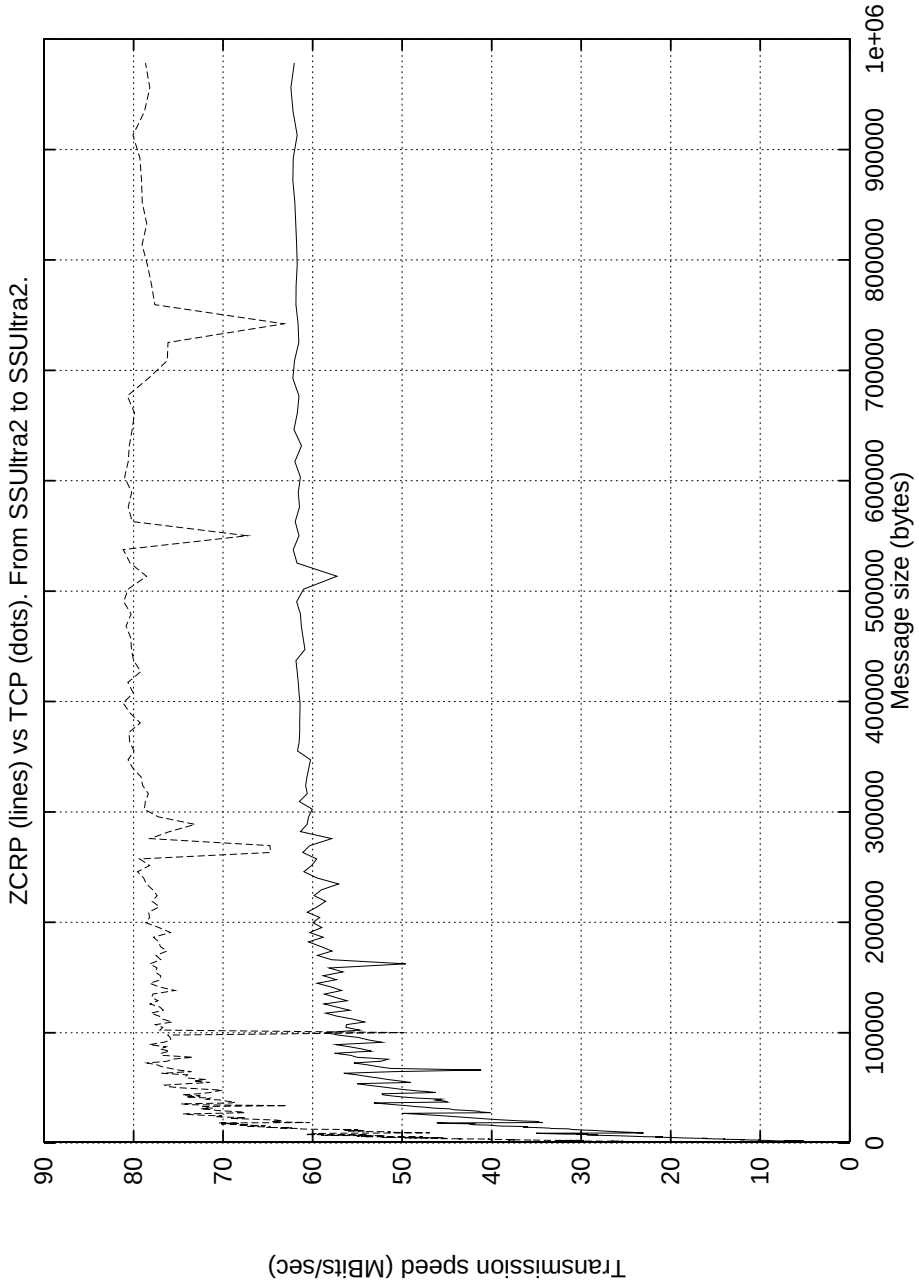

Figure 26: From SparcUltra2 to SparcUltra2.

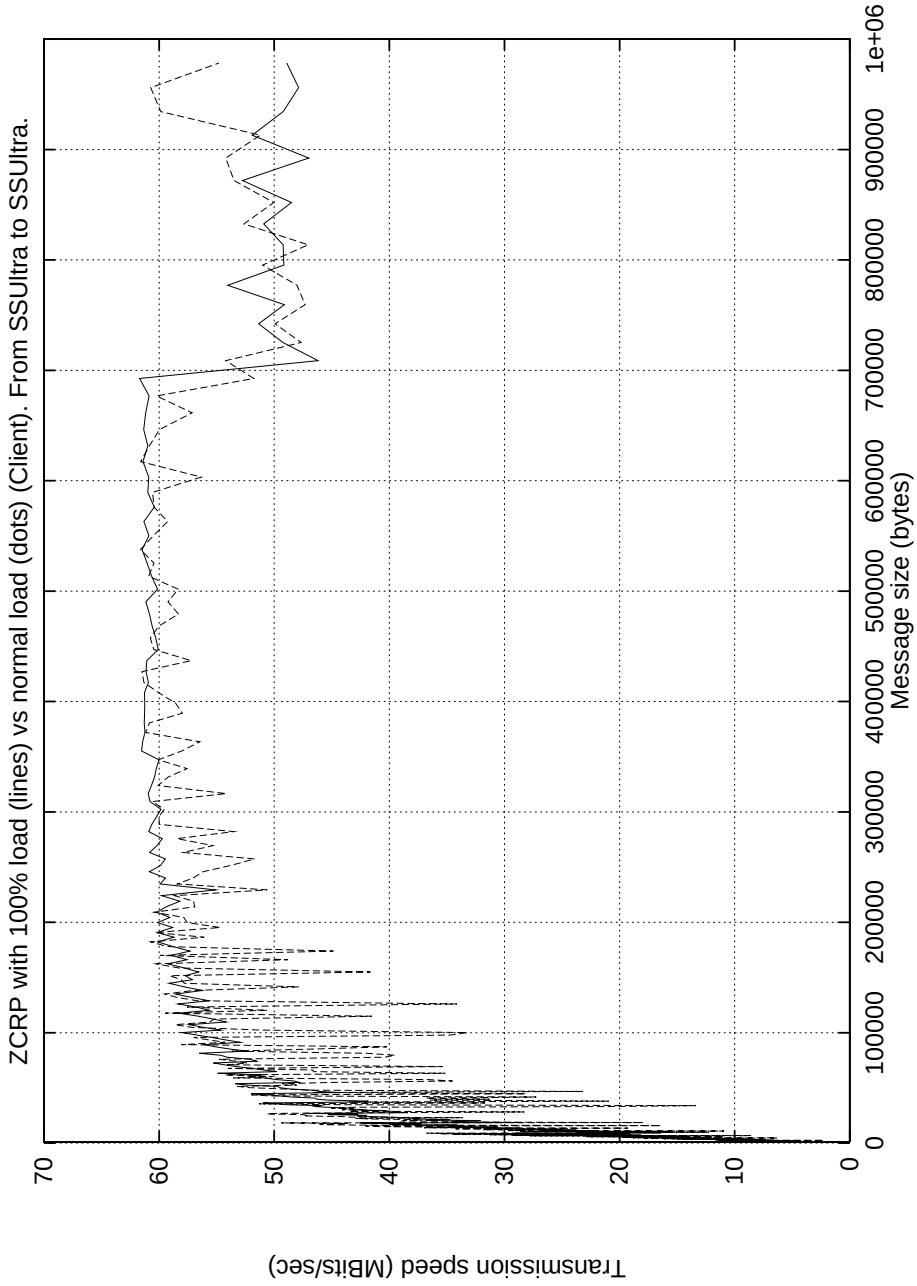

Figure 27: ZCRP: SparcUltra to SparcUltra with and without load.

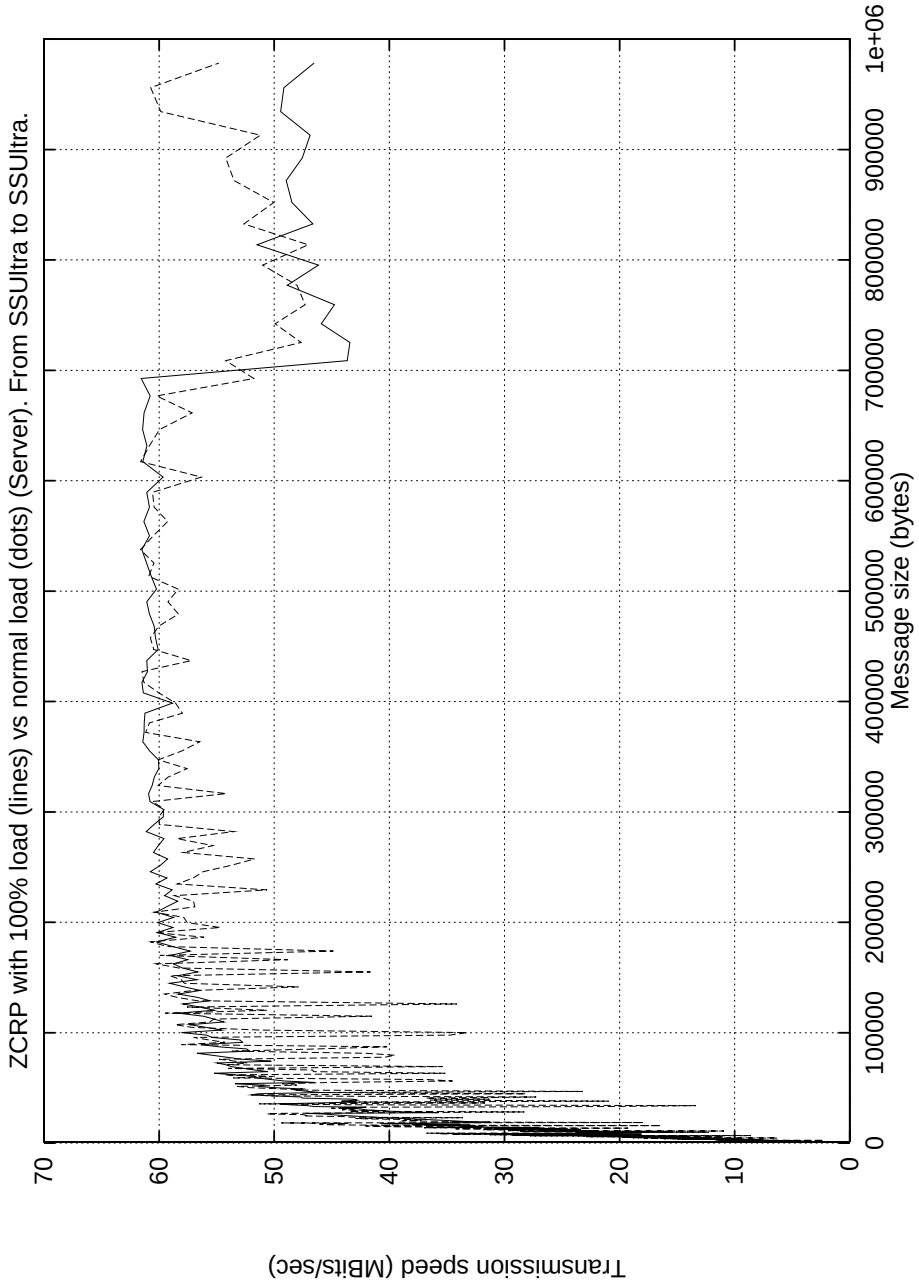

Figure 28: ZCRP: SparcUltra to SparcUltra with and without load.

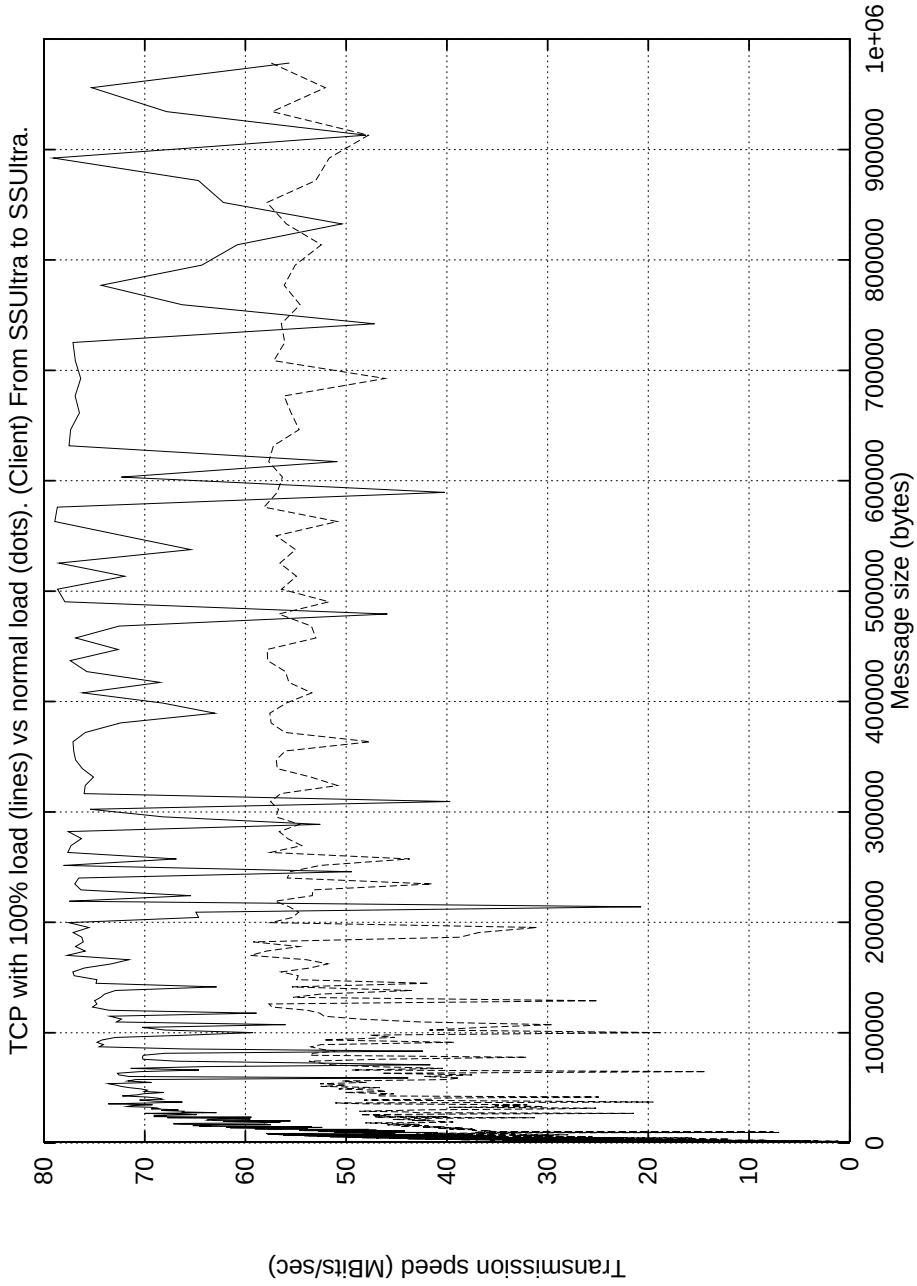

Figure 29: SparcUltra to SparcUltra with and without load.

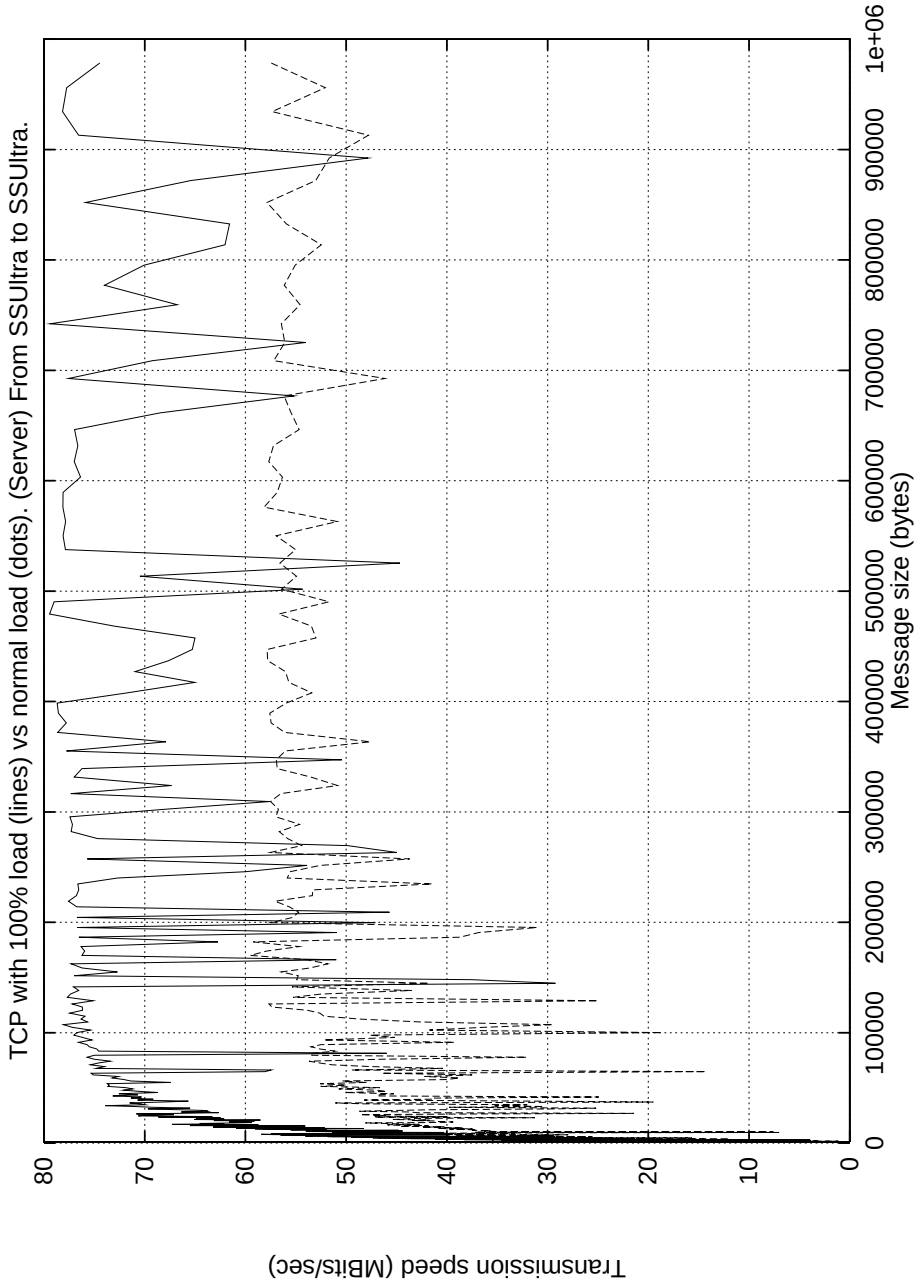

Figure 30: SparcUltra to SparcUltra with and without load.