Locking Effects in Multiprocessor Implementations of Protocols

Mats Björkman

Dept. of Computer Systems, Uppsala University
P.O. Box 325, S-751 05 Uppsala, SWEDEN

Email: matsb@docs.uu.se

Per Gunningberg

Swedish Institute of Computer Science
P.O. Box 1263, S-164 28 Kista, SWEDEN

Email: per@sics.se

To appear at SIGCOMM '93

Abstract

We investigate how to exploit shared memory multiprocessors for parallel protocol processing. We present a multiprocessor implementation of the x-kernel protocol environment from tha University of Arizona. A "processor-per-message" paradigm is used to partition the work over processors. Locks are used to protect shared protocol state and data. Mutual exclusion by locking can be costly if the parallel protocol code frequently accesses shared state and data. This paper addresses the effect of locking on performance.

The x-kernel is implemented in user address space on a Sequent Symmetry. Performance results of UDP/IP and TCP/IP are given, and we compare them to performance predictions from simulations. The measured speedups for the parallel implementations relative to the sequential ones are more than 12 times for UDP and 3 times for TCP. These limits are set by locking and machine contention.

1 Introduction

Communication can be a serious bottleneck in high performance distributed computing, where large amounts of data are moved between systems. As networks proceed to higher speeds, the performance bottleneck is shifting from the bandwidth of the transmission media to the capacity of hosts to process incoming and outgoing messages. Many hosts of today consist of a multiprocessor system with a shared memory and a small number of processors with their own caches. Having such a multiprocessor system, a number of the available processors may be dedicated to protocol processing. Protocol implementations in general do not exploit this parallel processing potential.

We have implemented a shared memory multiprocessor protocol execution environment, based on the x-kernel from the University of Arizona [HuPe91]. In this environment, we have implemented parallel versions of the protocols TCP, UDP and IP. The parallel x-kernel is implemented as a user space server on a 26 processor Sequent Symmetry.

We have chosen a parallelization paradigm which we will refer to as *processor-per-message*. It means that any processor in an array of processors can process the whole protocol stack for one message. Hence, an incoming or outgoing message is processed on the next available processor and several processors may execute in parallel on different messages belonging to the same connection. Our aim with this approach is to increase the throughput, the number of messages processed per second, rather than to decrease the latency, the processing time per message.

We present performance results for UDP/IP and TCP/IP. The measured speedups for the parallel implementations relative to the sequential implementations are more than 12 times for UDP and 3 times for TCP. These limits are set by locking of shared data structures and contention for the bus and memory.

Although our main measurements are performed using a no-copy driver, measurements including copying from internal buffers to network buffers and vice versa also show a speedup of more than 3 times for TCP, although with lower absolute message rates because of the additional processing time required.

Processors that execute communication software in parallel often need access to a shared state. For instance, if they process different messages on the same connection, they need access to the status information for this connection. Locking is the normal way to ensure mutual exclusion in a shared memory system. When processors compete for locks, waiting occurs, and this affects performance.

The two main performance limiting factors are locking and machine contention. Locking limits are mainly dependent on the protocol implementation, and not on the machine it executes on. Machine contention (contention for shared resources such as buses or memory) on the other hand, is machine dependent.

In this paper, we study the effects of locking on protocol processing performance. We investigate if multiprocessor protocol processing can be cost-effective with respect to locking, and if locking is the performance bottleneck in parallel protocol execution.

In order to predict locking effects on performance, we have developed a model and an algorithm for performance predictions. The algorithm describes the effects of locking on performance, and a simulation model based on the algorithm has been implemented and is used to predict performance.

Our measurements show that UDP performance is limited by machine contention and TCP performance is limited by locking. The difference is due to the fact that TCP has more state that needs to be protected from simultaneous accesses.

In chapter 2 we discuss efficient protocol processing and different parallelization paradigms with their suitability for multiprocessor implementation, together with references to related work. Chapter 3 concerns locking, and presents the locking model, our algorithm and the simulator. Chapter 4 describes the parallel x-kernel and the parallel protocols. Chapter 5 presents simulation results for TCP/IP and UDP/IP. Chapter 6 presents performance figures from the real parallel implementation measurements, and compares them to the simulation results. In Chapter 7 conclusions are drawn and further work suggested.

2 Efficient Protocol Processing

There exist several proposals on special hardware and software for reducing the processing time for messages. Of main interest for our approach are the software methods since they may work together with our "processor-per-message" paradigm. The overall goal with these methods is to reduce the number of instructions executed for a message.

Protocol processing can be divided into two parts, control functions and data manipulation/movement functions. The data manipulation functions consist of two or three phases. First there is a read phase of data, then a manipulation phase followed by a write phase for some functions. For very simple data manipulation functions, such as checksumming, byte alignment, etcetera, the time to read and write to primary memory dominates the processing time on fast RISC architectures. In the control part there are functions for header and connection state processing. Jacobson et al have demonstrated that the control part of TCP/IP processing can match gigabit network performance for the most common size of messages with appropriate implementations [CJRS89]. However, data manipulation functions present a bottleneck.

In general, our paradigm can benefit from shorter control processing times. The speedup factor for the paradigm depends on the proportion of locked code to unlocked code plus two or three instructions to set and reset locks. If this proportion does not change the speedup factor will remain the same.

Executing data manipulation functions in a shared memory multiprocessor means that the data part of messages must be copied from the main memory to the cache of a processor. The amount of data moved will be the same as for a uniprocessor. Therefore, the performance will most likey be limited by the capacity of the bus and memory system. When we introduced a data movement function in our parallel TCP, we achieved the same relative speedup as without. The approach also seems to work well with the method Integrated Layer Processing [ClTe90, GPSV91]. This method reduces the number of read and write phases to primary memory. In principle, all manipulations will be done within one read and one write. There is even a potential synergy effect compared to the uniprocessor case, since the bus can be used when a processor is busy with manipulations. However, to demonstrate this is left for further work.

2.1 Parallel protocol processing

Parallel protocol processing is interesting for multiprocessor systems demanding high communications bandwidth. A key factor for efficiency in shared memory systems is caching. A shared memory multiprocessor system typically uses caching extensively to avoid memory contention. Cache consistency protocols ensure that cache contents are consistent throughout the system.

Another factor that affects performance is the level of granularity at which the processing is parallelized. Passing of control or data between processors causes overhead. A coarse parallelization grain lowers this overhead. The granularity effects on performance will be discussed in the sections describing the different parallelization paradigms below.

2.1.1 Processor-per-message

The processor-per-message paradigm has been presented in the introduction. We have chosen this paradigm for several reasons: First, it is a suitable choice for high performance communication since a small number of heavily used connections can be supported efficiently. Second, it gives a coarse-grain parallelism (message-sized grains) suitable for shared memory multiprocessor implementation. Third, it gives flexibility in the number of processors allocated to communication. Fourth, it gives an implicit balancing of processor load, since messages are distributed evenly over the processors.

There are also disadvantages with this paradigm. Connection state has to be shared among the processors in order to be able to process messages belonging to the same connection in parallel. Synchronization of accesses to the shared state may involve mutually exclusive code regions or critical data protected by locks.

An example of another system using this paradigm is the Raven Protocol Framework from University of British Columbia [GoNe92]. In Raven, manipulations of shared connection state are isolated to one block in each protocol, the Connection Control Block (CCB). In contrast to our approach, this means that protocols with complex connection state may need major rewriting. The CCB is accessed in serial by all processors, while all the other code in the protocol can be executed in parallel. Protocols with a large connection state, notably TCP, will have large CCBs that limit parallelism. In Raven, OSI protocol stacks have been implemented and performance improvements have been shown for up to 4 processors [GNI92].

Another example is the system by Jain et al from Columbia University [JSB90]. They propose an architecture using the processor-per-message paradigm with an array of message processing processors, and separate processors for low level processing and host interfacing. In contrast to our approach, they use special-purpose processors and the interconnection of processors is via buses dedicated to message processing traffic.

There are other paradigms for parallelizing protocol execution. Below follows a brief presentation of the most common ones. References to these paradigms can also be found in for example [GNI92].

2.1.2 Processor-per-connection

Processor-per-connection means that every connection is dedicated to one processor. If there are more connections than available processors, connections share processors. For each connection, one processor executes the whole protocol stack.

A problem with the protocol-per-connection paradigm is how to handle multiplexing. At multiplexing points, streams of messages are split or joined. If several higher-layer connections are multiplexed onto one lower-layer connection in the middle of the protocol stack, all of these connections must be handled by the same processor.

An advantage over the processor-per-message paradigm is that protocol state do not need to be shared between processors. The main disadvantage is that processor-per-connection suffers from fragmentation of processor resources, i.e. a bursty connection which saturates one processor cannot be offloaded by other processors in the system.

2.1.3 Processor-per-protocol

Processor-per-protocol means that each processor is responsible for executing a separate protocol. Messages are passed from processor to processor on their way up or down the protocol stack. Each processor performs protocol processing for one, or possibly several, specific protocols for each message. Having separate processors for incoming and outgoing traffic means that incoming and outgoing traffic do not interfere, but protocol state has to be shared between processors, removing one of the main advantages with this paradigm.

We believe that there are several disadvantages with this paradigm. Message processing will be pipelined at the speed of the slowest protocol, and the granularity for parallel execution is often too small for protocols with short execution times.

2.1.4 Processor-per-function

Processor-per-function means that each processor performs a specific function within a protocol or a function common to more than one protocol. This is a more fine-grained paradigm than processor-per-protocol, and has the additional advantage that functions may be performed in parallel. We think this paradigm is useful for special-purpose hardware implementation of processors.

The main disadvantages are that granularity will be even smaller than in processor-per-protocol, that protocol state must be shared between processors, and that there will be parallel access to messages.

Examples of systems using this paradigm are those reported by Zitterbart et al [Zit89] and Giarrizzo et al [GKWW89]. Zitterbart and Giarrizzo both use the processor-per-function paradigm, combined with processor-per-protocol parallelism. They implement the protocol handling on Transputer arrays.

3 Modelling parallelism and locking effects

In this section we will discuss locks, describe a model for parallel and cyclically running communication software and give definitions for predicting the effects of locking on performance.

If message processing on each processor is completely independent, the number of processed messages will scale up linearly with the number of processors until it reaches the capacity limit of the machine. Unfortunately, independent processing is not the normal case. Messages which must share the same communication hardware or software, such as a common protocol stack or a connection state, are not independent of each other. For correct processing only one processor at a time may use a shared resource. For example, the data structure holding the current sequence number in a protocol must be atomically updated. Since processors may have to wait until shared resources are available, the parallel processing speedup will not scale linearly.

3.1 Spin locks

We use *locks* in the message processing code in order to protect shared common data structures. A lock is granted to only one requesting processor at a time. It is typically a reserved variable which is guaranteed by the system to be updated atomically. A region of sequential code which has updates to shared data structures is protected by locks. A processor which is about to enter a protected region first checks whether another processor is executing in the region by reading the lock variable. If the region is free, the variable is set or *locked* and the region is entered. At exit, the variable is reset or *unlocked*. When processors reach a locked region, they must wait until the lock holding processor exits. Depending on system support, a handful of instructions are executed in order to lock a lock, including an atomic "test-and-set" instruction or equivalent.

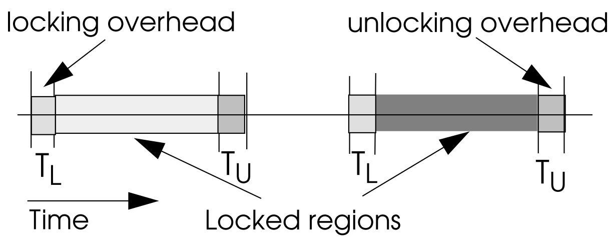

Figure 1 Locking overhead.

Spin locks are used in shared memory multiprocessor systems with local data and instruction caches. This means that waiting processors "spin" in a loop in which they continuously test the lock. There are two arguments for this kind of non-productive use of the processors. First, spinning processors do not disturb other processors because they are "spinning in the cache". Second, for short locking times it may be inefficient to do a time-consuming context switch to another process.

3.1.1 Spinning effect on performance

During spinning, the processors will not do any productive work. How will this reduce the effective parallelism? Assume that the time to process a message is C on a single processor. Then the number of processed messages per second, the *rate*, is $\frac{1}{C}$ for a processor. For a multiprocessor using N processors and with **no** spinning or other contention effects, the rate will be $N\frac{1}{C}$. When spinning occurs, the message processing time will be prolonged with spinning periods. Assume that the average sum of all spinning periods is Ω, then the rate is reduced to $N\frac{1}{C+\Omega}$.The spinning period Ω is a stochastic function of N, such that the probability of spinning will increase with N. Intuitively, when more and more processors contend for the same shared resources the probability for spinning will increase. Other factors that affect this probability of spinning are: 1) the proportion and distribution of locked code versus unlocked code and 2) the relative start and speed of the processing on each processor. If we can find Ω as a function of N and the other two factors, then we can calculate an expectation value for the performance increase of parallel processing with respect to locking.

3.2 Cyclically Running Code

We believe that a cyclic model of the execution time for message processing is useful for predicting locking effects. This model is based on Clark and others [CJRS89, ThWo92] observation that one execution path in the communication software is often dominating, the so called "common path". (Van Jacobson's header prediction algorithm exploits this anticipated behavior.) This path represents the normal case when there are no exceptions and thus we will assume that it is possible to predict the execution time of the path. This path will be repeated in a cyclic fashion as long as arriving messages follow the same path. For example, this is a common situation for bulk transfers. The message processing rate will therefore be dominated by the path and we will only model locking in this part of the software.

In a multiprocessor, all processors may process messages simultaneously in the common path. A processor will pick a message from a queue and start to process it. When a processor is finished, it will immediately pick the next message in a queue and run the "common path" all over again. As long as there is a queue of messages all the processors will show a cyclic behaviour.

We will assume a perfect cyclic behaviour in the sense that each cycle will take the same amount of time when there are no spinning periods. We also assume that there is always a queue of messages waiting to be processed. The only thing that will change the cycle length is spinning. Obviously, in a real implementation, such a stable situation can only be maintained for shorter periods.

If the duration of each spinning period is deterministic, it will be possible to predict the performance effects of spinning. We assume that the processors will be granted the locks in the same order they request them in order to ensure a deterministic behaviour. This request granting policy will also ensure fairness between the processors. If two or more processors request a lock exactly at the same clock tick, we assume a global priority between them.

In figure 2 we have illustrated the locking interactions between two processors. Initially there is a *starting skew* between processors 1 and 2, which in this figure means that processor 2 will start it's cycles at a later point in time. As can be seen in the figure, the effect of spinning on a processor is that forthcoming phases of the cycle on that processor will be delayed with the length of the spinning period. As a consequence, future locking periods may be in conflict with another set of locking periods on other processors, which in turn will delay other phases on other processors, and so on.

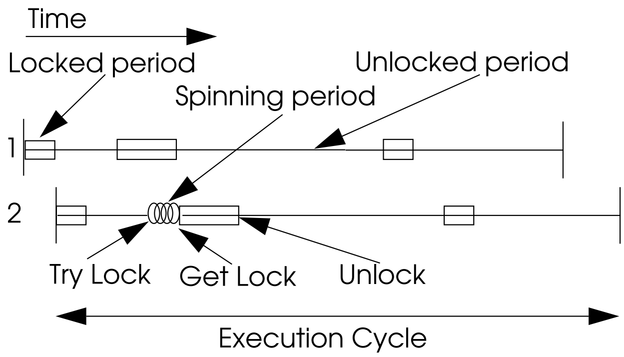

Figure 2 Locking in parallel execution.

It is our observation that the cycles in the beginning of the parallel execution are successively stretched out until a stable situation occurs. By a stable situation we mean the situation when cycle $k+m$ is identical to cycle k for all processors. The average length of the cycles at a stable situation can be used to predict performance. In figure 2, a stable situation has been reached after the first cycle.

3.2.1 Upper limit on number of processors

Since only one processor at a time can hold a lock, the sum of all locking periods in a cycle puts a limit on the degree of parallelism. The upper limit on the speedup factor from the parallelism is $\frac{C}{\Delta}$, where Δ is the sum of all locked periods. It should be observed that this is an upper limit, achievable only for the most favourable distribution of locks (no locking conflicts at a stable situation and Ω=0, for N < $\frac{C}{\Delta}$). Figure 3 shows the upper limit on the speedup and what may be achieved for a typical distribution. (Ω > 0 for all N).

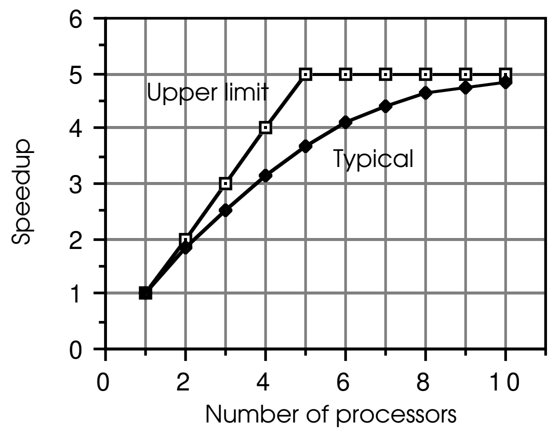

Figure 3 Limit on parallelism.

In figure 4 we illustrate what happens when locking time dominates and the performance increase stagnates. In this example, the cycle has only one locking period, and the locking time is one third of the total cycle time.

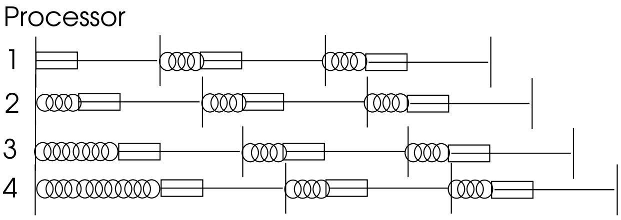

Figure 4 Locking time is dominating and performance has stagnated. A stable situation is reached.

3.3 Predicting cycle lengths

We will now present how the length of the cycles can be predicted with our cyclic model. The cycle is divided into phases. A phase consists of three periods, first a *spinning period*, then a *locking period* and a *non-locking period*. The length of the locking periods and non-locking periods will remain the same on all processors in all cycles while the spinning periods will vary.

Then the length of a cycle k, **including spinning** at processor i can be formulated as: $\sum_{j=1}^{M} (\varepsilon_{ikj} + \delta_j + \beta_j)$, where:

M: the number of phases in a cycle

ε_{ikj}: *spinning* period of processor i, in cycle k and phase j.

δ_j : *locking* period in phase j.

β_j : *non-locking* period in phase j.

A phase j in cycle k on processor i has a starting time which we will denote t_{ikj}. The first phase of the first cycle starts at t_{i11} after a skewing time s_i. The start of a phase $j+1$ can then be formulated as (illustrated in figure 5):

$$t_{i11} = s_i$$
$$t_{ik(j+1)} = t_{ikj} + \varepsilon_{ikj} + \delta_j + \beta_j \qquad \text{for } j<M$$
$$t_{i(k+1)1} = t_{ikj} + \varepsilon_{ikj} + \delta_j + \beta_j \qquad \text{for } j = M$$

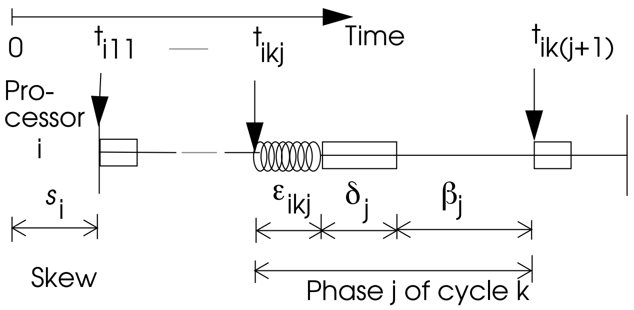

Figure 5 A phase in a cycle.

Now, spinning periods will occur at t_{ikj} with the length ε_{ikj}. This length depends on the state of all other processors, i.e. if they are in the locking period or are waiting to enter a locked region. If ε_{ikj} can be derived, then the starting point for the next phase, $t_{ik(j+1)}$ for processor i can be predicted. However the state is only known for the processor with the smallest t_{ikj}. This can be formulated in the following iterative algorithm:

1. Chose the processor i with smallest t_{ikj} = *time_now*;
2. **if**< lock is free> **then** $\varepsilon_{ikj} = 0$;
   elseif <proc. *n* has lock> **and** <no waiting proc.>
   then ε_{ikj} = <time at unlock> - *time_now*;
   else $\varepsilon_{ikj} = \sum_{WaitingProcessors} LockingPeriods(i) +$
   (<time at unlock> -*time_now*);
3. Compute $t_{ik(j+1)}$

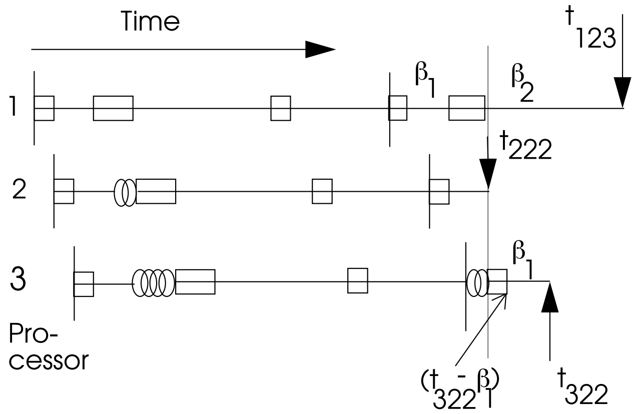

Figure 6 The spinning period for processor 2, ε_{222} can be calculated. For processor 2 and phase 2, there is a locking conflict with processor 3, which means processor 2 has to spin to the end of the locking period of processor 3, hence $\varepsilon_{222} = (t_{322} - \beta_1) - t_{222}$.

By following the algorithm, we will get the starting time t_{ikj} for all phases and processors and can calculate an expected cycle length. In figure 6, spinning for processor 2 can be calculated.

3.4 Simulation tool

We have designed a simulation tool which is based on the above algorithm. It simulates the parallel execution until it finds a stable cyclic situation for a given lock distribution in a cycle and for a given number of processors. The simulator will vary the skew (the relative starting point of each processor), and for each skew combination the length of the stable cycle is recorded. A stable cycle is found by the following procedure: At the end of each cycle, the state of all processors is recorded and compared to the previously recorded states. When a match is found, a stable cycle has been reached. The output from the simulator are cycle lengths and their relative frequency. The time in the simulator is measured in *ticks*. The basic tick length used in our simulations is 1 microsecond.

3.5 Initial skew selection

For a small number of processors and ticks it is possible to check all combinations of initial skews. However, it does not take a math professor to see a combinatory explosion lurking for larger systems.

For larger systems we instead select a random set of skew points, uncorrelated to locking times. Results from simulations will then be statistical approximations. Experiments have shown that these approximations are very accurate. For random sets of 1000 skew points, the standard deviation, relative to the correct result from an exhaustive simulation, was less than 1 percent for the sample distribution in figure 7. The number of random sets is a parameter by which the user can control the execution time and the accuracy of the results.

3.6 Simulation output

The output from a simulation is for each number of processors the length of stable cycles and their frequency, ie how many of the skew combinations that will produce the same stable length. In figure 7 we present in a diagram form the output from a simulation of the distribution in figure 6. The simulator has run this distribution for 2 to 5 processors. We have divided the cycle into 18 ticks, of which 4 are locked. As the horizontal axis we use the cycle length in number of ticks, and on the vertical we give the frequency for this length. As expected, the frequency distribution will move towards longer cycle lengths as the number of processors increase.

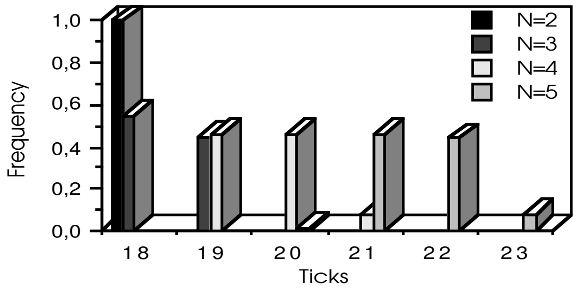

Figure 7 Distribution of cycle lengths for 2 to 5 processors.

Note that the results in figure 7 show that for 4 processors there exists no stable cycle with 18 ticks. Since 4 out of 18 ticks are locked, the ideal speedup could mislead us to believe that stable cycles with no spinning periods exists. However, the distribution used in this example is a good example of a distribution where the achievable speedup is lower than the ideal speedup for 4 processors.

From the frequency distribution we can calculate the expected cycle length by summing the frequency-lenght product. For the distribution in figure 7 and 5 processors it is 21.62. This can be interpreted as the expected mean value from a real implementation when the skew is unpredictable.

4 Parallel x-kernel

In this chapter, we present our parallel implementation of the x-kernel from University of Arizona [HuPe91]. The parallel x-kernel is implemented as a user space server on a 26 processor Sequent Symmetry running Dynix V3.1.1. Wherever needed in the rest of the paper, the original x-kernel will be called the *sequential* x-kernel, and the implementation presented will be called the *parallel* x-kernel.

4.1 The x-kernel

The (sequential) x-kernel is a protocol execution environment developed at University of Arizona [HuPe91]. It is a self-contained execution environment in which protocols can be implemented in a modular and uniform, yet efficient, way. The x-kernel provides the protocol implementer with a set of uniform interfaces to other protocols and to operating system types of services likely to be used by a protocol implementer. The x-kernel also contains a set of tools that interface x-kernel services to those of the host operating system. The x-kernel is implemented on a number of different machines and operating systems. Both kernel and user space implementations are available. Below we will highlight the main features of the x-kernel (based on the α-release of x-kernel v3.2).

The Uniform Protocol Interface (UPI) provides a protocol implementation with a uniform way of calling other protocols. With the UPI, implementers do not need to be aware of which protocols run on top of or below the protocol being implemented. Rather, by using a dynamic graph, the binding of protocols into hierarchies is deferred until linking time. Top and bottom protocols are called *anchor protocols*, and anchor the x-kernel protocols in the host environment. Typical anchor protocols are application programming interfaces and device drivers.

The x-kernel also provides the implementer with a set of tools that implements commonly used routines in a machine-independent or portable way. The most important tools are; the Message Tool, implementing efficient handling of messages, the Process Tool, implementing thread creation and handling, the Event Tool, allowing routines to be queued for future execution, and the Idmap Tool, mapping external identifiers to internal.

4.2 Parallelization of the x-kernel

For reasons explained above, we decided to base the parallel x-kernel on the processor-per-message paradigm. In the parallel implementation, each processor is assigned a direction, inbound or outbound. This gives higher cache hit rates for protocol code, when executing in the common path. The anchor protocols are implemented on separate processors as well as the event handler (see below) In figure 8, each box represents a processor.

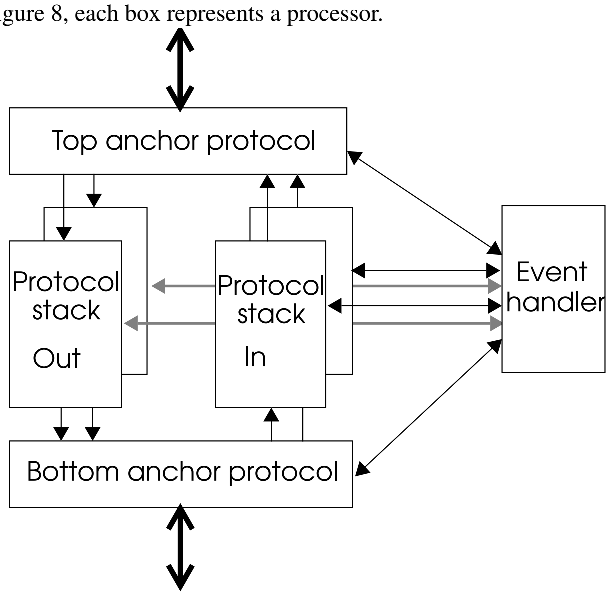

Figure 8 The parallel x-kernel architecture.

The parallelization work was divided into two phases; parallelization of the x-kernel itself, and parallelization of the protocols TCP, UDP, IP and an x-kernel Ethernet anchor protocol. In the x-kernel itself, the three major tasks were parallelization of thread execution, memory handling and event handling.

The sequential x-kernel has a single master lock. The lock is locked when an x-kernel thread starts, and released when it exits. An executing thread is guaranteed to run to completion without being preempted. In order to allow parallel execution of threads, the master lock was replaced by smaller and shorter locks that protect shared state and data.

Since dynamic allocation of shared memory is very time consuming in the Dynix environment, fast memory allocation routines for the most common types of dynamic data was implemented using pre-allocated pools of shared memory. Data buffers, queue entries and x-kernel message structures are examples of shared data types with their own allocation routines.

Even though we have chosen the processor-per-message paradigm for reasons above, we use a dedicated processor as an event handler server. There are several reasons for this: First, event handling is asynchronous to its nature, and, if performed on a message processing processor, it might upset the real-time behaviour of the communication. Second, event handling is very suitable for implementation on a separate processor, since little interaction takes place between the event handler and its clients. Third, event handling and its associated clock interrupts can be costly, and a separate event handling processor would reduce the load on protocol processing processors.

4.2.1 Locking in protocols and in the x-kernel

Locks are used to protect shared state that must be atomically updated. There are two extreme strategies as to where to perform locking. One is to put all locks in the protocol code, the other is to put all locks in the x-kernel code. Putting all locks in the x-kernel code causes too much overhead for locking control and deadlock prevention [Hut92]. The other extreme, to put all locks in the protocol code, means that the protocol implementer must be aware of the internal details of the x-kernel, and also that locking times for complex x-kernel operations become very long. The strategy adopted in our parallel implementation is to let the x-kernel protect the environment state, and the protocol implementer is responsible for protecting the protocol and protocol implementation state.

We divide the lock types into three classes: *protocol specific*, *protocol implementation specific* and *environment specific*. Protocol specific locks protect the state of the protocol. An example is the TCP window state. Protocol implementation specific locks are used for data structures in a specific implementation. An example from this implementation is the TCP Pcb (Protocol control block) list. The environment specific locks protect structures in the execution environment. An example is the x-kernel Object state. Table 1 lists the major lock types for each class..

Lock type class	Major lock types
Protocol specific	TCP send window, TCP receive window, TCP round trip time
Protocol implementation specific	TCP Timer, TCP Sendbuffer Queue, TCP Pcb, TCP Reassembly
Environment specific	x-kernel Object, x-kernel Message, x-kernel Event, x-kernel Queue, x-kernel Memory allocate, x-kernel Idmap, Scheduling

Table 1 Lock types in protocol and x-kernel environment.

When introducing locks, we were cautious not to introduce deadlock potentials. The approach we use is to make sure that the x-kernel never calls protocol code while holding locks. Furthermore, x-kernel code never calls x-kernel code while holding locks if there is a risk for indirect recursion. Left to the protocol implementer is to beware of calling x-kernel code while holding locks if she does not know that the x-kernel code will not call protocol code. The deadlock potential with multiple locks in a complex protocol cannot be ignored. We plan to implement a tool that analyzes possible call sequences and finds potential deadlocks.

The IP, UDP and anchor protocol code never lock more than one protocol lock at a time. For TCP, however, the complex state required multiple locks.

5 Multiprocessor Simulations

We have done performance simulations with respect to locking for the UDP/IP/Ethernet and TCP/IP/Ethernet protocol stacks. We have simulated both outbound and inbound sides for both stacks, and the performance results are very similar. This is consistent with the observations of [CJSR89]. Therefore, we have chosen to present figures for outbound sides only. Throughout the rest of this paper, we will sometimes use the names UDP and TCP, respectively, as shorthand for the whole stacks.

5.1 Simulation results from UDP and TCP traces

We have collected execution traces on the Sequent Symmetry from the UDP and TCP stacks. The traces are depicted in figure 9. The traces are time-stamps from one cycle on one undisturbed processor. Time-stamps were obtained by reading the system clock at

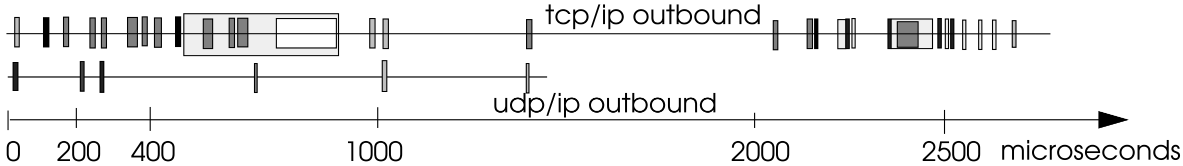

Figure 9 Locking sequences in TCP/IP and UDP/IP.

locking and unlocking operations, as well as at the start and end of the cycle. The granularity of the clock is one microsecond. In this section we will describe the traces and how we used them in simulations to predict performance. Results from simulations were used to calculate the speedups for 2 to N processors relative to the execution time for 1 processor. In all simulations, 1000 random skew points were used. Mean execution times were used when calculating speedups. Standard deviations were small (typically less than 5%), and are therefore not shown in diagrams.

5.1.1 UDP simulations

The outbound trace for UDP/IP is depicted in figure 9. As can be seen from the figure, the locks are generally short and sparsely distributed. An upper estimate of the achievable speedup limit with respect to locking is obtained by dividing the cycle length with the locking time, which gives about 30 times.

We used the traces from UDP/IP and ran the simulator up to 20 processors. In diagram 10, the simulation results are presented.

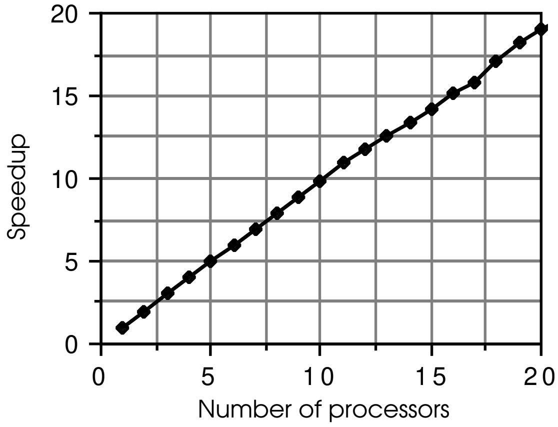

Figure 10 Simulation results for UDP, outbound direction.

As expected, the performance increases almost linearly with the number of processors.

5.1.2 TCP simulations

Traces from TCP/IP are also depicted in figure 9. TCP has a more complex locking scheme than UDP and there are several locks inside locks. Simulation results from this trace can be found in diagram 11. The performance increase starts to level off at 7 processors.

6 Multiprocessor measurements

Multiprocessor measurements were collected on the parallel x-kernel implementation described in chapter 4, running as a user process on a 26 processor Sequent Symmetry S81. Each processor is rated at 2.7 mips. We had 20 of the processors available for the protocol processing array.

We have measured the throughput of the parallel x-kernel implementation for the protocol stacks UDP/IP and TCP/IP and the x-kernel simulated Ethernet protocol. Since the Sequent Dynix Ethernet communication has a maximum throughput of around 500 Kbytes/s, it quickly became the bottleneck, and was therefore not used in the measurements. Instead, a high performance network driver

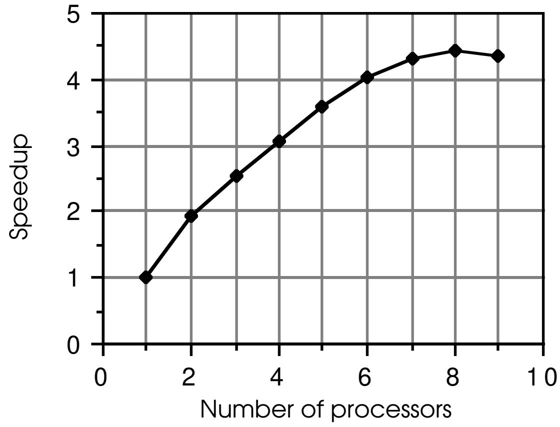

Figure 11 Simulation results for TCP, outbound direction.

was simulated by inserting a sink and a source in the x-kernel Ethernet code. The sink flushes outgoing data, the source creates data and supplies it to the x-kernel as incoming frames. Apart from the sink and the source, the protocol implementations are interoperable with sequential TCP and UDP implementations.

6.1 UDP measurements

We measured the throughput of UDP/IP/Ethernet with 1Kbyte user data messages. Simulations indicate that locking effects in the UDP/IP/Ethernet stack would not saturate the system until more than 20 parallel protocol processors were used.

The first measurements were made using the simulated network driver with one frame copying, simulating copying from x-kernel buffers to network buffers. As shown in figure 12, measurements with this driver reached a saturation point at 7 processors, at about 11 Mbit/s. The bottleneck is data copying in the bottom anchor processor.

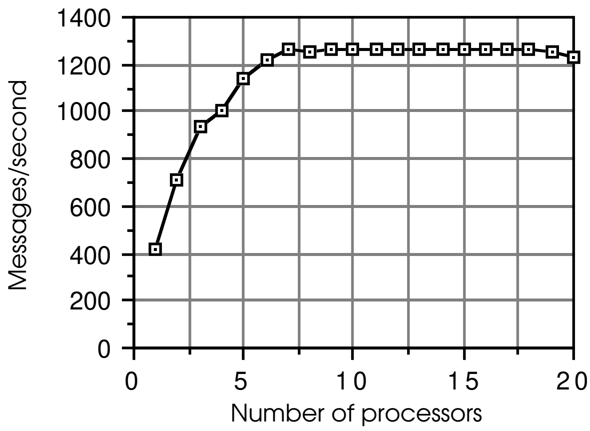

Figure 12 Performance of UDP/IP/Ethernet with copying.

The driver was modified not to copy data. This simulates a network interface that reads or writes data directly from/to x-kernel buffers. Figure 13 shows the performance results for this setup. The system saturates at about 11 processors. The bottleneck here is the anchor processors, working (at saturation point) at a message rate about 11 times that of each of the protocol processors.

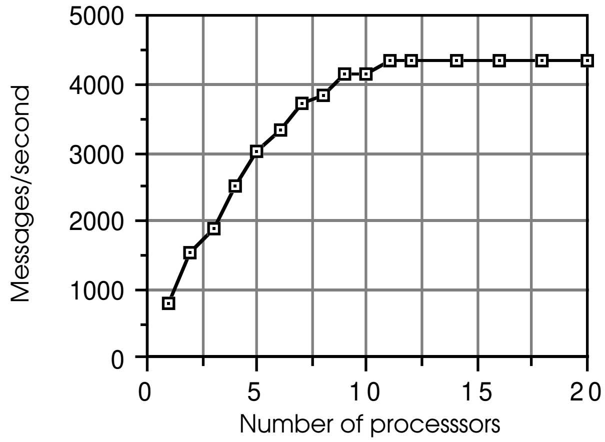

Figure 13 Performance of UDP/IP with no-copy driver.

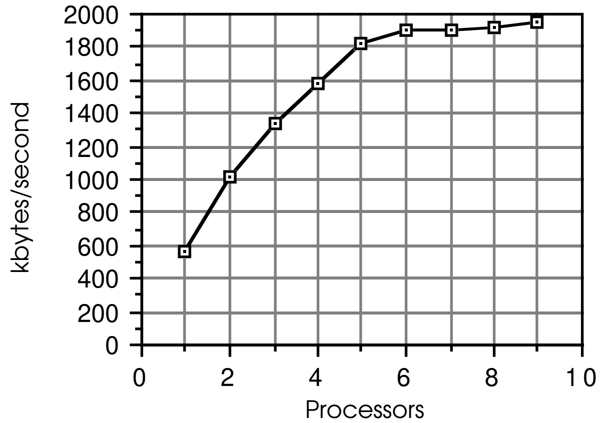

Figure 15 Measured performance of TCP/IP/Ethernet.

In our next experiment, we removed the anchor processors by letting the protocol processors themselves be the sinks and sources of the messages. This mode tests the performance limits of the protocol processor array.

Figure 14 shows the performance of the UDP stack in array testing mode. The protocol processors do not suffer from saturation in the range of 2 to 20 protocol processors.

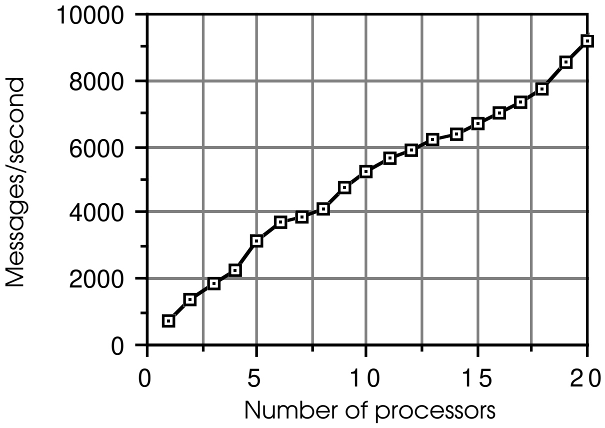

Figure 14 Performance of UDP/IP in array testing mode.

6.2 TCP measurements

The TCP protocol has a more complex state than UDP, and simulations indicate that locking saturation should occur at 7 parallel protocol processors. In contrast to the TCP implementation in the sequential x-kernel distribution, the parallel TCP implementation was improved by adding the one-behind Pcb cache and header prediction as in Unix BSD Network release 2.

Since TCP is a connection-oriented protocol and expects sequenced packets and acknowledgements, simulating network interfaces was not as straightforward as in the case of UDP. Our solution was to generate incoming messages in the order they were expected to arrive, and to generate acknowledgements accordingly.

We did not use TCP checksumming. In order to be consistent with simulations, acknowledgements were handled on the same processors as data. We measured TCP in steady state, where both the Pcb cache and the header prediction hit. Figure 15 show performance results for the TCP stack with the no-copy driver. Measurements including copying from x-kernel buffers to network buffers exhibit a similar speedup, although the absolute data rate is about 30 percent lower, because of the additional processor time required for the copying.

6.3 Correlation with simulation results

In this section, simulated and measured performance figures for UDP and TCP are compared. The reasons behind the noted discrepancy is discussed in section 6.4.

6.3.1 UDP

As seen in figure 16, the measured performance of UDP does not increase with the same factor as the simulations predict. Measured performance increases with a factor of about .7 per processor.

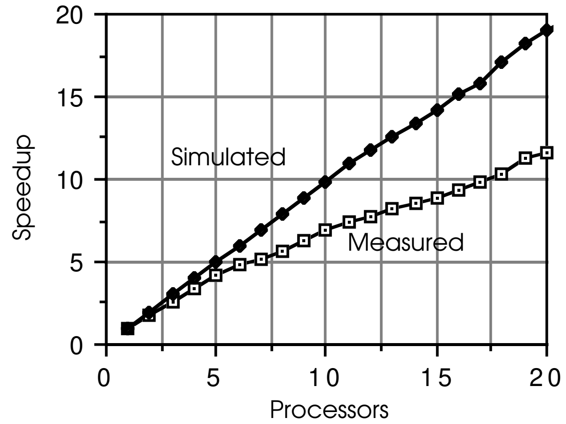

Figure 16 Comparison of simulated and measured speedup for UDP.

6.3.2 TCP

Figure 17 shows measurement results for TCP relative to simulations. Like in the case of UDP, the performance increase is smaller than the simulation results.

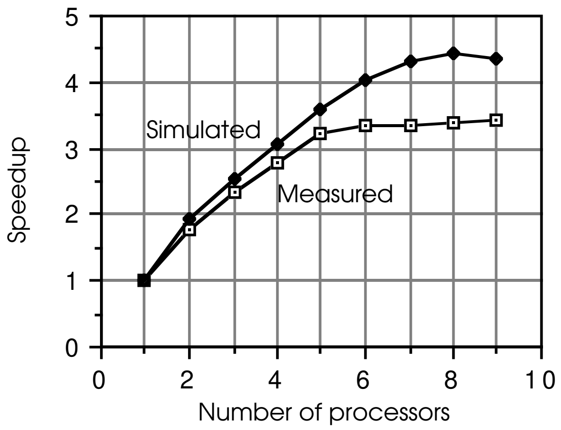

Figure 17 Comparison of simulated and measured speedup for TCP.

6.4 Discrepancy observations

For both UDP and TCP, the speedup is smaller than predicted. The difference is roughly proportional to the number of processors, about 55 µs per processor in the case of UDP, 120 µs in the case of TCP. The main reason behind the observed discrepancy is machine dependent contention for the buses and memory.

We are currently developing a queueing model that includes machine contention as well as locking. Figure 18 shows results from the new model overlaid on the TCP graphs from figure 17. A description of the model can be found in [Bjö93].

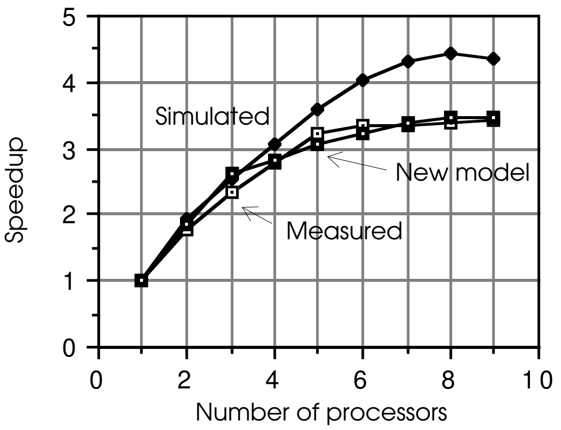

Figure 18 New model including machine contention compared to simulations and measurements for TCP/IP.

7 Conclusions and directions for further work

From the presentation we conclude that locking is one of several performance-affecting factors when executing protocols in parallel on a shared-memory multiprocessor system. For protocols with little state, as UDP and IP, the cost of locking shared state is small and is not the limitation of parallelism. Instead, the limit is machine dependent contention to shared buses and memory. For protocols with more complex state, as TCP in our implementation, locking puts a limit on parallelism. In our implementation, locking saturates the system at 6 parallel processors.

7.1 Further work

We are currently studying the effects of machine dependent contention on parallel execution of protocols. We are developing a model describing the compound effects on performance caused by locking and machine dependent contention. Preliminary results can be found in [Bjö93].

Many protocols in use today were not originally designed for parallel execution, e.g. TCP, which has a complex protocol state. It would be interesting to investigate the suitability of other existing transport protocols for parallel implementation, and to find the effects parallel implementation would or should have on transport protocol design. It would also be interesting to make a TCP implementation parallel from scratch and compare its performance to sequential implementations.

8 References

[Bjö93] M. Björkman, *Lock and Memory Contention Effects in Multiprocessor Implementations of Protocols*, Dept. of Computer Systems, Uppsala University.

[CJRS89] D. Clark, V. Jacobson, J. Romkey, H. Salwen, *An Analysis of TCP Processing Overhead*, IEEE Communications Magazine, June 1989.

[ClTe90] D. Clark, D. Tennenhouse, *Architectural Considerations for a New Generation of Protocols,* Proc. ACM SIGCOMM 1990.

[GKWW89] D. Giarrizzo, M. Kaiserswerth, T. Wicki, R. Williamson, *High-Speed Parallel Protocol Implementation*, Proc. 1st IFIP Workshop on Protocols for High-Speed Networks, Zürich 1989.

[GNI92] M. Goldberg, G. Neufeld, M. Ito, *A Parallel Approach to OSI Connection-Oriented Protocols*, Proc. 3rd IFIP Workshop on Protocols for High-Speed Networks, Stockholm 1992.

[GoNe92] M. Goldberg, G. Neufeld, *The Raven Protocol Framework*, Technical report, University of British Columbia, Vancouver.

[GPSV91] P. Gunningberg, C. Partridge, T. Sirotkin, B. Victor, *Delayed Evaluation of Gigabit Protocols,* Proc. 2nd MultiG Workshop, Stockholm 1991.

[HuPe91] N. Hutchinson, L. Peterson, *The x-kernel: An Architecture for Implementing Network Protocols*, IEEE Trans. on Software Engineering, Jan. 1991.

[Hut92] N. Hutchinson, *Protocols vs. Parallelism*, Proc. from the x-kernel Workshop 1992, Dept. of Computer Science, University of Arizona.

[JSB90] N. Jain, M. Schwartz, T. Bashkow, *Transport Protocols Processing at GBPS Rates*, Proc. ACM SIGCOMM, 1990.

[Zit89] M. Zitterbart, *High-Speed Protocol Implementations based on a Multiprocessor-Architecture*, Proc. 1st IFIP Workshop on Protocols for High-Speed Networks, Zürich 1989.