Software-Based Communication Latency Hiding for Commodity Workstation Networks

Volker Strumpen
Department of Electrical and Computer Engineering,
University of Iowa, Iowa City, IA 52242
strumpen@eng.uiowa.edu

Abstract *A variety of latency hiding techniques has been investigated at the hardware level. However, except multithreading, which may require substantial program structuring effort, other software-based latency hiding methods have not been investigated. In this paper, we consider design alternatives for latency hiding other than multithreading. Furthermore, we present experimental evidence for the validity of a new technique for software-based communication latency hiding for commodity workstation networks: Up to 80 percent of useful computational power can be squeezed out of a workstation CPU while communicating with TCP/IP via an Ethernet and almost 90 percent while communicating across the Internet.*

1 Introduction

Computer networks are becoming a primary compute resource for engineers and scientists. Technological advances and the wide-spread availability of these cost-effective systems are making them attractive for daily use. Although CPU speed and memory capacity of current workstations can exceed supercomputer node performance, interconnect technology limits the use of clusters for parallel computing to coarse grained applications. Furthermore, clusters are usually shared by users, and the unpredictable changes of CPU load and network load complicate their efficient utilization for distributed parallel computing.

Current research is emerging to level off the differences between multiprocessors and workstation networks. Research prototypes such as *T [21] or HPAM [18] based on the Medusa host-network interface [2] provide hardware support for efficient latency hiding implementations. However, since current commodity workstations and commodity networks do not offer architectural support for communication latency hiding, we consider software-based latency hiding techniques. The only widely accepted software-based mechanism for hiding latencies in workstation networks is multithreading, which may require substantial program (re-)structuring. For many single-threaded applications, especially data-parallel and SPMD style programs, asynchronous message passing as offered by Intel's NX message passing interface with non-blocking `isend` and `irecv` operations and the corresponding blocking synchronization operation `msgwait`, suffices to program communication latency hiding. We discuss design alternatives for implementing communication latency hiding with asynchronous message passing and present an implementation that provides latency hiding in an effective and portable manner.

2 Communication Latency Hiding

Latency hiding techniques have been considered since computers are built. As far back as 1946, Arthur Burks, Herman Goldstine and John von Neumann wrote [6]: "Using Ac [Accumulator] for this purpose eliminates the possibility of computing and reading from or writing on the wires simultaneously. However, simultaneous operation of the computer and the input-output organ requires temporary storage and introduces a synchronizing problem, and hence it is not being considered for the first model." Recently, when the goal of scalable parallel computing arose, latency hiding became a topic of wide-spread interest.

2.1 Latency Hiding Techniques

Different designs for latency hiding have been considered at the hardware and software level. Among the trend-setting hardware designs are the following developments: The HEP architecture [23] introduced *multithreading* to increase processor utilization. Later, memory latency hiding techniques gained popularity when investigated for distributed shared memory architectures: *Prefetching* [11], *coherent caches* [11] and *relaxed memory consistency* [10] are known mainly from Stanford's DASH project [17]. The third approach hides communication latencies by means of additional communication hardware (processors) in distributed memory systems such as *T [21] or the Intel

Paragon. Now, processor designers are introducing *non-blocking caches* together with dynamic instruction scheduling in next-generation microprocessors such as the MIPS R10000 [20] to overcome the penalty of cache misses.

Multithreading has not only been investigated at the hardware but also at the software level, in particular for distributed memory systems. Hiding communication by means of multithreading is usually understood as executing some thread, ready for useful computation when another thread blocks in some operation, for example a communication operation. Multithreading is now widely accepted as a means for hiding latencies, not only for memory load operations but also for hiding interprocessor communication latencies [5, 7, 12, 13, 15].

For multicomputers, multithreading and asynchronous message passing, such as offered by Intel's NX interface [22], are widely used latency hiding mechanisms. Nevertheless, they require careful structuring of parallel programs. With multithreading the programmer must identify independent threads of control that can be executed concurrently. With asynchronous message passing, a non-blocking `send` or `recv` operation and the corresponding synchronization call (`msgwait` in NX) are usually separated by as much useful computation as possible to allow for worst case message delays and desynchronization of the communicating processes. Since compiler support for both techniques multithreading and asynchronous message passing is still in its infancy, programmers accept the additional programming effort in order to improve scalability.

2.2 Software-based Latency Hiding

To implement communication latency hiding in computer networks with commodity workstations and commodity networks that do not offer hardware support for latency hiding a concise understanding of the overheads introduced by a software solution is necessary. In the following, we shed light on the mechanisms that enable commodity workstations to provide for communication latency hiding.

Multithreading on commodity workstations suffers from two major obstacles: (1) A context switch is relatively expensive, and (2) for communication, the processor has to perform protocol computation, which can lead to extensive switching between the threads of a user process and the kernel. Since context switching is relatively expensive, this may render multithreading counterproductive. Furthermore, in many cases such as data parallel programs and SPMD style programs, multithreading seems to be overkill; using asynchronous message passing appears to be simpler, because it might not require restructuring of the whole program but only a different positioning of the communication primitives. Therefore, in the following, alternative software-based implementations of communication latency hiding for asynchronous communication are considered.

Figure 1 illustrates three design alternatives for implementing latency hiding in a portable manner, i.e. on top of existing operating systems. The basic idea of these implementations is to interleave useful computation of an application with protocol computation necessary for interprocess communication. This scheme simulates asynchronous message passing, and works effectively with a single thread of control. However, there is no restriction as to support multithreaded applications as well with these designs. In fact, multithreading is used as an implementation technique for the designs shown in Figure 1(b) and 1(c).

Two-process setting: Besides the application process, a separate process handles communication across the network. This two-process setting is implemented in PVM [9], where the PVM daemon `pvmd` is responsible for communication. Scheduling of the application process and communication process is done by the operating system's scheduler.

Kernel threads: The application process consists of one (or more) calculation thread(s) (calc) and a communication thread (com). These threads are kernel threads, i.e. the threads are managed within the kernel, which in particular handles the scheduling of these threads within the user's address space.

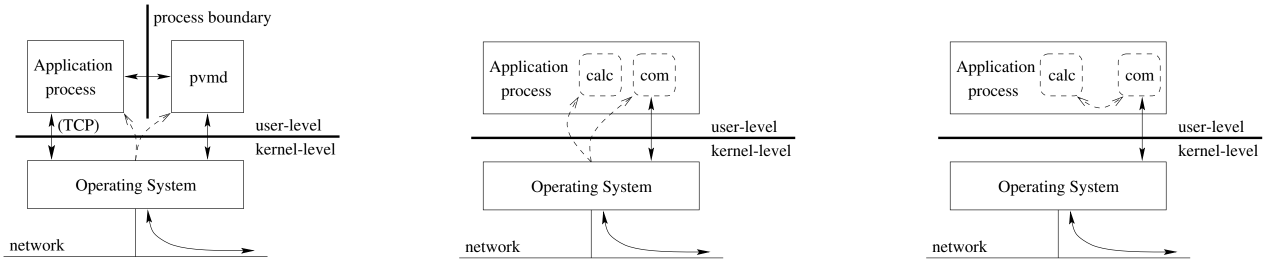
User-level threads: As with kernel threads, the application process consists of one (or more) calculation threads (calc) and a communication thread (com). However, these threads are user-level threads, and scheduling of the threads is implemented within user space.

These design alternatives are discussed in the following.

2.2.1 Two-process Setting

Figure 1(a) illustrates the two-process setting as implemented in the popular PVM system [9], which provides most runtime services by means of the PVM daemon `pvmd` running on all machines of the configuration. Messages are exchanged between two application processes by copying the data across the process boundary into and out of `pvmd`, and communicating the message across the network by means of the UDP/IP protocol. Reliable communication is implemented within `pvmd` by retransmitting lost UDP packets and resolving reordering of packets.

More recent versions of PVM avoid the copy between application process and `pvmd` and establish a TCP connection directly between two communicating application processes (cf. line attributed with (TCP) in Figure 1(a)). This implementation delivers raw TCP performance. The performance gain of not crossing the process boundary is up



(a) Two processes (PVM) (b) Kernel threads (c) User-level threads

Figure 1: Design alternatives for portable interprocess communication with latency hiding. Data paths are indicated by solid lines, scheduling control paths by broken lines. Protection boundaries are marked by bold lines.

to a factor of 3–4. However, in general, a TCP connection does not allow for communication latency hiding if blocking communication primitives are used in a single-threaded application. Latency hiding could be implemented with non-blocking communication primitives in principle. However, augmenting the application code with non-blocking communication retries is not recommendable. Placing them into an application independent module, executed as process or thread, is the only clean design alternative.

The original PVM implementation, based on UDP, exhibits one advantage: The operating system's scheduler schedules application process and `pvmd`. Apparently, this avoids the need for a scheduler that handles the interleaving of these two processes. The author implemented a communication module [25] that exploits the kernel scheduler. To avoid the overhead of copying messages between the application process and communication process (`pvmd` in Figure 1(a)), messages are exchanged via shared memory segments. This implementation delivered reasonable CPU utilization when communicating via the Internet [25]. However, for faster networks such as an Ethernet, the time slices of the kernel scheduler are too long to allow for effective interleaving of application process and communication process. Thus, the problem with this design alternative is that the kernel scheduler does not support the scheduling of processes based on system call behavior, in particular blocking reads and writes on sockets, as would be necessary for sensible interleaving of computation and communication.

2.2.2 Kernel Threads

Figure 1(b) shows the use of kernel threads for communication latency hiding. In principle, kernel threads overcome the scheduling problem of the two-process setting. The kernel can schedule the kernel threads of an application, because the communication protocol runs in the kernel, and therefore, the kernel has the information about protocol idle times or blocked system calls. Consequently, the kernel thread scheduler can switch to another kernel thread, if there is no network activity. For such-the-like purposes *scheduler activations* [1] have been proposed.

Compared to the two-process setting, kernel threads avoid the crossing of the process boundary and thus the overhead of message copying. However, similar to the two-process setting the user-kernel boundary needs to be crossed for both message copying and thread scheduling.

Besides these performance disadvantages due to protection, another practical aspect affects kernel thread implementations from the runtime system developer's point of view: portability. For example, Sun's SunOS release 4.1 offers kernel threads as so-called light weight processes. However, context switching as a consequence of blocking system calls is not implemented[1].

2.2.3 User-level Threads

Figure 1(c) presents the use of user-level threads for implementing communication latency hiding. The first key advantage of user-level threads compared to the two-process setting and kernel threads is that the user-kernel boundary is not crossed for scheduling. However, this in itself presents two disadvantages. First, an additional user-level scheduler is required. Second, no information about the kernel's protocol state is available to the user-level scheduler. However, the latter point becomes void when lifting the communication protocol entirely into user-space is considered. In this case, the user-level implementation is (almost) independent of the operating system and no scheduling support is needed from the operating system.

The rest of this article describes an implementation of latency hiding by means of user-level threads on top of the UNIX operating system. As the experimental results show, such an implementation suffices for today's commodity networks. In particular, CPU utilization reaches

[1] See SunOS man page: "Because there is no kernel support for threads at present, system calls effectively block the entire pod [collection of threads sharing an address space]."

$80\,\%$ – $90\,\%$ on Ethernet and Internet. However, for future high-performance networks, lifting the communication protocol into user-space is the natural step towards delivering the bandwidth of such networks [19, 26].

3 Implementation Issues

This section describes the design of a user-level implementation for communication latency hiding. Since a single-threaded application is a special case of a multithreaded application with respect to our scheduling policy, multithreaded applications are treated as the general case.

The implementation of communication latency hiding is based on the distinction of application threads, marshal threads and a single communication thread per process. The communication thread multiplexes outgoing messages and demultiplexes incoming messages of the entire application process. Optionally, each message is associated with a unique marshal thread that marshals or unmarshals message fragments that are communicated by the communication thread, and converts data representations if necessary.

The communication thread manages message transfers by means orthogonally linked lists of data structures containing the state of a message. A message identifier is used to look up the corresponding data structure on receipt by means of hashing. Linked lists are used to distinguish all messages of a process with respect to their transfer states. With these data structures, the communication thread is able to multiplex all messages of a process across all virtual (TCP) connections independent of the state of individual application threads. Within the communication thread message passing is based on non-blocking system calls. If a message fragment cannot be launched into or pulled out of the kernel buffer after a predefined number of retries, another active socket is selected in a round-robin fashion. If none of the sockets are ready after a predefined number of sweeps across all sockets, the communication thread switches back to an application thread.

The overlapping of calculation and communication is implemented by interleaving application threads, marshal threads and the communication thread. By integrating the communication thread into the thread scheduling policy superfluous context switches are avoided, and a sensible polling of the network is possible. The goal of the scheduling policy is to minimize the runtime of the communication thread and maximize the time a single or multiple application threads are scheduled into the CPU. If messages are pending, the communication thread is preemptively scheduled after a predefined timeout. In the current implementation this timeout equals the resolution time of the alarm timer; on Sun SPARCstations the resolution is $20\,ms$.

Figure 2 illustrates the interleaving of an application thread, a marshal thread and the communication thread

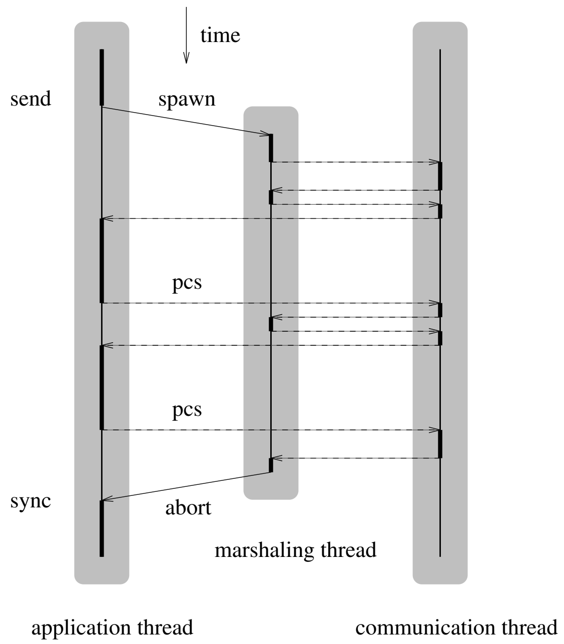


Figure 2: Interleaved message handling in one process consisting of three threads: An *application thread* that sends a message using the message-specific *marshal thread*, and a process-specific *communication thread* that interfaces to TCP/IP. Broken lines indicate context switches, and bold lines mark the running thread.

when sending a message. Note, that the marshal thread is optional, depending on whether linearization of data structures or conversion of data representations is necessary. During the send operation, a marshal thread is spawned. After a fragment has been marshaled into the bounded runtime buffer, a non-preemptive context switch to the communication thread is performed. If the fragment has been delivered successfully to the operating system, the communication thread returns to the marshal thread to pack the next fragment. The communication thread switches back to the application thread, if the transport layer (kernel) refuses to accept any further bytes. If the message is still pending, the context is preemptively switched (pcs) back to the communication thread after the timeout. The network can recover or transfer messages of other processes while the application thread is running. When the entire message is consumed by the runtime layer, the marshal thread is terminated and control is returned to the application thread.

4 Experimental Results

In order to measure the latency hiding capabilities of our implementation and the whole underlying system a synthetic benchmark has been created. Using an application

```
int sem;

void contf(void)
{
   volatile sem = 0;
}

void
bench(int pid, margv_t *sma, margv_t *rma)
{
   int smid, rmid, dc=0;

   sem = 1;
   rmid = recv(pid,0,unmarshal,rma);
   smid = send(pid,0,marshal,sma);

   mcont(smid,contf,0);

   while (sem)
      fdaxpy(), dc++;
   printf("dc   %d"\n, dc);

   msync(smid);
   msync(rmid);
}
```

Figure 3: Benchmark for communication latency hiding.

for this purpose would impose another, algorithm specific limitation to the maximum gain due to communication latency hiding. Our benchmark avoids the problems of intermixing the application specific and system specific effects and discloses the latency hiding potential of the underlying system. Application performance is reported in [24, 25].

Figure 3 shows the code fragment that contains the kernel of the benchmark. Two processes are executing this code and simultaneously exchange two messages of equal size. The communication thread's protocol ensures that this is possible without deadlock. We measure the amount of useful computation that is done within the overlap region. The BLAS `daxpy` ($\vec{y} = a\vec{x} + \vec{y}$) [16] has been chosen as a well known computational kernel to benchmark the CPU power available for useful computation. It is applied to double precision vectors of length 1000. The daxpy computation is scheduled within a `while`-loop that is terminated by means of a message continuation [24]. A message continuation is a computation that is executed as soon as the corresponding message transfer is completed. The resulting count of daxpy executions is used as the measure of useful CPU utilization overlapping the communication. We define *latency hiding efficiency* E_{lh} as:

$$E_{lh} = \frac{\text{number of daxpy executions for latency hiding}}{\text{number of dedicated daxpy executions}}.$$

In the following, E_{lh} is given in per cent.

A semaphore variable `sem` is used to terminate the daxpy loop by switching its value within the message continuation `contf`. Within the C function `fdaxpy` the original FORTRAN routine is called. Storage for vectors $\vec{x}$ and $\vec{y}$ is allocated elsewhere.

Table 1 and Table 2 list the results of this benchmark for a LAN configuration consisting of two Sun SPARCstation10 connected via Ethernet. The payloads of the simultaneously exchanged messages varies from 4 KByte to 1 MByte. The transfer times correspond to the case without running the daxpy loop and without message continuations. The dedicated daxpy counts are measured with a different program that contains the daxpy loop which is interrupted by means of an alarm signal after a period of time equal to the transfer time. The send-daxpy counts are generated with the benchmark program given above, and the receive-daxpy counts with the message continuation parameterized with the receive message (`rmid`).

The difference between Table 1 and Table 2 is the application of marshaling: For the measurements presented in Table 1, an array of size payload is transferred without marshaling (`marshal` = `unmarshal` = `(void *) NULL`). In contrast, the values shown in Table 2 are generated with marshaling. Here arrays of double precision numbers of size payload are marshaled by copying each double precision value into and out of the message. This gives a notion of the overhead for marshaling dynamic data structures. No conversion of data representations is performed, which would just introduce further marshaling overhead.

These measurements demonstrate the viability of the runtime layer approach as an additional layer on top of the TCP/IP suite in order to improve CPU utilization. Note that the communication times of the experiments with latency hiding are equal to the listed transfer times without overlapping calculation within measurement accuracy. These transfer times represent socket-based TCP/IP performance. The differences between the send-triggered and receive-triggered versions can be explained with the interleaving of the message fragments on the bus network. Depending on which of the two messages is actually transferred first, these differences vary slightly from measurement to measurement.

The benchmark version without marshaling delivers latency efficiencies larger than 70 $\%$. That means the CPU spends more than 70 $\%$ of the communication time with useful computation, and not with TCP protocol computation or thread scheduling. The program with marshaling employs the CPU additionally for copying the array elements into and out of a message buffer. Despite of the loss of CPU cycles for useful computation, more than 50 percent of the CPU is utilized in the LAN configuration.

It should be noted in this context, that all measurements where obtained in a normal working environment. It was just taken care that no other CPU intensive processes were

Payload [KByte]	Transfer [ms]	dedicated daxpy	daxpy (send)	E_{lh} [%] (send)	daxpy (recv)	E_{lh} [%] (recv)
4	10	40	34	85	37	93
8	19	62	56	90	56	90
16	31	86	66	77	64	74
32	61	153	121	79	108	71
64	118	275	199	72	225	82
128	233	545	392	72	413	76
256	463	1070	784	73	749	70
512	931	2140	1444	67	1522	71
1024	1849	4200	2978	71	2992	71

Table 1: Latency hiding benchmark in a LAN: Two SPARCstation10 connected via Ethernet, **without marshaling** of messages.

Payload [KByte]	Transfer [ms]	dedicated daxpy	daxpy (send)	E_{lh} [%] (send)	daxpy (recv)	E_{lh} [%] (recv)
4	12	44	32	73	31	70
8	20	62	28	45	59	95
16	34	91	58	64	90	99
32	66	160	87	54	111	69
64	122	297	152	51	177	60
128	236	550	293	53	283	51
256	461	1070	573	54	582	54
512	931	2148	1134	53	1107	52
1024	1878	4290	2246	52	2163	50

Table 2: Latency hiding benchmark in a LAN: Two SPARCstation10 connected via Ethernet, **with marshaling** of messages (copying of single double values).

running during the experiments. Furthermore, the bandwidth of the Ethernet is almost optimally utilized. For instance, according to Table 1, two 1 MByte messages are exchanged via the Ethernet within 1.849 seconds, which corresponds to a throughput of 9.1 Mbit/s.

Compared to the variance of the timings in our Ethernet LAN, measurements in the Internet are often not reproducible, and timings may easily vary by a factor of two. Therefore, the latency hiding potential of the Internet is measured with a single benchmark run, executing a (send) message continuation, and measuring the dedicated daxpy loop count with respect to the transfer times of this benchmark program. Table 3 lists the results. Experiments with a receive continuation yielded comparable values. Because the latency of the Internet is roughly one order of magnitude bigger than that of a 10 Mbit/s Ethernet, a higher CPU utilization rate can be expected. Not surprisingly, the daxpy counts for large messages achieve rates close to 90 percent.

The numbers given in the tables above can be used to assess the computational work that should be provided within the overlap region of an application to fully exploit the latency hiding feature. Consider the example of a matrix-matrix multiplication. Multiplying two quadratic matrices of order n requires n^2 `daxpy` executions. Thus, while exchanging two 1 MByte messages across the Internet, we can calculate about 6 percent of a matrix-matrix multiplication of order $n = 1000$ (daxpy counts were measured with vectors of length 1000!). In the faster Ethernet, 0.2 to 0.3 percent of the multiplication can be completed. These numbers can be used to tune the partitioning of an application and optimize the number of processors used [24].

5 Related Work

The results presented here outperform those of a previous implementation of a latency hiding protocol of the author described in [25]. That work is based on a two-process setting, one acting as a communication process, the other as calculation process. Messages are exchanged between these processes via shared memory and transferred across processors by means of the communication processes. This scheme is relatively simple to implement. The scheduling

Payload [KByte]	Transfer [ms]	dedicated daxpy	daxpy (send)	E_{lh} [%]
4	391	661	410	62
8	471	717	477	67
16	1180	1829	637	35
32	1362	2108	1677	80
64	4275	7665	5639	74
128	6021	10860	9158	84
256	11613	20556	17753	86
512	24869	40452	35429	88
1024	39660	71818	61525	86

Table 3: Latency hiding benchmark in the Internet: One SPARCstation10 at ETH Zürich and one SPARCstation2 at MIT, Cambridge. Send-daxpy counts **without marshaling** are measured on the SPARCstation10.

task is performed by the native operating system scheduler. The process switches induce a higher overhead than the thread based implementation described here. Both implementations differ with respect to their latency hiding performance. Internet experiments are about 10 percent more efficient with the user-level thread implementation, and Ethernet latency hiding efficiency improves from about 60 percent up to 80 percent.

Several projects have been emerging recently to ameliorate the problems of traditional message passing style programming. Among these are generic models such as active messages [27], and those aiming at better performance and/or easing parallel programming by means of message-driven execution models. More closely related to our approach of communication latency hiding are Charm [14] and Cilk [4].

Charm [14] introduced the message-driven programming style. It provides automatic mapping and scheduling. So-called chares are programmed by providing entry points for messages. Messages are received by the Chare kernel and buffered in a message queue. A queue manager picks a message according to a predefined load balancing strategy, and starts execution of the code associated with the entry point carried by the message. Message reception and sending use explicit buffering to provide non-blocking communication. However, the UDP based communication protocol of the networking implementation does not overlap communication with useful computation.

Cilk [4] focuses on efficient thread scheduling. Its main contribution is a work-stealing scheduler fulfilling theoretical efficiency criteria that are also matched in practice. The network implementation is based on Phish [3], a predecessor of Cilk. Phish implements split-phase operations based on UDP. This scheme allows for useful computation between a communication request and the corresponding reply, but not for overlapping the communication itself.

The Distributed Filaments [8] approach applies multithreading for overlapping communication of pages and calculation in a user-level distributed shared memory system. This implementation is based on a split-phase operation to handle a remote page fault. After a request is sent off the local scheduler switches to another thread while waiting for the requested page. Like Charm and Phish, distributed filaments implements a reliable transport protocol on top of UDP. In [8], results of overlapping communication and computation are reported. The gain of 21 % is relatively low compared to the numbers presented in the paper at hand. It is not clear, whether this value of gain is a consequence of the application structure or due to the fact that user-level multithreading is used for overlapping.

6 Conclusions

Design alternatives for software-based communication latency hiding in commodity workstation networks have been discussed. Furthermore, a portable implementation based on user-level threads is introduced. This implementation utilizes idle times of the underlying TCP/IP protocol layers for useful computation while communicating data. We provide experimental evidence to show that there is sufficient idle time in these layers on commodity hardware to overlap communication almost entirely (70–90 percent), if payloads larger than 4 KByte are transferred. How much of this potential can be utilized depends on the algorithmic structure of the application. The presented technique of latency hiding works with single-threaded as well as multithreaded applications. Thus, we can overlap computation and communication in single-threaded applications, and do not suffer from scheduling overhead due to multithreading.

Acknowledgement

Thanks to Peter Arbenz, Tom Casavant, Walter Gander, Charles Leiserson, Balkrishna Ramkumar and Rolf Strebel.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] D. Banks and M. Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[3] R. D. Blumofe and D. S. Park. Scheduling large-scale parallel computations on networks of workstations. In *3rd International Symposium on High-Performance Distributed Computing*, pages 96–105, San Francisco, California, August 1994.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[5] B. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *19th Annual International Symposium on Computer Architecture*, pages 214–223, Queensland, Australia, May 1992.

[6] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. In C. G. Bell and A. Newell, editors, *Computer Structures: Readings and Examples*, chapter 4, pages 92–119. McGraw-Hill, 1971. This article is published originally as a report prepared for the U.S. Army Ordnance Department, 1946, and in A. H. Traub, editor, "Collected Works of John von Neumann," volume 5, pages 34–79, The Macmillan Company, New York, 1963.

[7] Edward W. Felten and Dylan McNamee. Improving the performance of message-passing applications by multithreading. In *Scalable High Performance Computing Conference*, pages 84–89, Williamsburg, Virginia, April 1992. IEEE.

[8] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First USENIX Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1994.

[10] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, Santa Clara, California, April 1991. ACM.

[11] Anoop Gupta, John L. Hennessy, Kourosh Gharachorloo, T. Mowry, and W. D. Weber. Computative evaluation of latency reducing and tolerating techniques. In *18th Annual International Symposium on Computer Architecture*, pages 254–263, Toronto, Canada, May 1991.

[12] Matthew Haines and Wim Böhm. An evaluation of software multithreading in a conventional distributed memory multiprocessor. In *5th IEEE Symposium on Parallel and Distributed Processing*, pages 106–113, Dallas, Texas, December 1993.

[13] John Holm, Antonio Lain, and Prithviraj Banerjee. Compilation of scientific programs into multithreaded and message driven computation. In *Scalable High-Performance Computing Conference*, pages 518–525, Knoxville, Tennessee, May 1994. IEEE.

[14] Laxmikant V. Kale. The Chare kernel parallel programming language and system. In *International Conference on Parallel Processing*, volume II, pages 17–25. IEEE, August 1990.

[15] K. Kurihara, D. Chaiken, and A. Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 91–101, April 1991.

[16] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[17] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[18] Richard P. Martin. HPAM: An active message layer for a network of HP workstations. In *Hot Interconnects II*, 1994.

[19] Ron Minnich, Dan Burns, and Frank Hady. The memory-integrated network interface. *IEEE Micro*, 15(1):11–20, February 1995.

[20] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual*, 2nd edition, June 1995.

[21] G. M. Papadopoulos, R. Greiner, and M. J. Beckerle. *T: Integrated building blocks for parallel computing. In *Supercomputing'93*, pages 624–635, Portland, Oregon, November 1993.

[22] P. Pierce. The NX Message Passing Interface. *Parallel Computing*, 20(4):463–480, April 1994.

[23] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE: Real-Time Signal Processing IV*, pages 241–248, San Diego, California, August 1981. Volume 298.

[24] Volker Strumpen. *The Network Machine*. PhD thesis, Department of Computer Science, Swiss Federal Institute of Technoloy (ETH) Zürich, 1995.

[25] Volker Strumpen and Thomas L. Casavant. Implementing Communication Latency Hiding in High-Latency Computer Networks. In *High-Performance Computing and Networking*, LNCS 919, pages 86–93. Springer-Verlag, Milano, Italy, May 1995.

[26] T. von Eicken, V. Avula, A. Basu, and V. Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, February 1995.

[27] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.