PULC: ParaStation User-Level Communication. Design and Overview

Joachim M. Blum and Thomas M. Warschko and Walter F. Tichy

University of Karlsruhe, Dept. of Informatics
Postfach 6980, D-76128 Karlsruhe, Germany
email: {blum,warschko,tichy}@ira.uka.de

Abstract. PULC is a user-level communication library for workstation clusters. PULC provides a multi-user, multi-programming communication library for user level communication on top of high-speed communication hardware. In this paper, we describe the design of the communication subsystem, a first implementation on top of the ParaStation communication card, and benchmark results of this first implementation. PULC removes the operating system from the communication path and offers a multi-process environment with user-space communication. Additionally, we have moved some operating system functionality to the user level to provide higher efficiency and flexibility. Message demultiplexing, protocol processing, hardware interfacing, and mutual exclusion of critical sections are all implemented in user-level. PULC offers the programmer multiple interfaces including TCP user-level sockets, MPI [CGH94], PVM [BDG+93], and Active Messages [CCHvE96]. Throughput and latency are close to the hardware performance (e.g., the TCP socket protocol has a latency of less than 9 μs).

Keywords: Workstation Cluster, Parallel and Distributed Computing, User-Level Communication, High-Speed Interconnects.

1 Introduction

Common network protocols are designed for general purpose communication. These protocols reside in the kernel of an operating system and are built to interact with divers communication hardware. To handle this diversity, many standardized layers exist. Each layer offers an interface through which the other layers can access its services. This layered architecture is useful for supporting divers hardware but leads to high and inefficient protocol stacks. Protocols which are using standardized interfaces of the operating system are unaware of superior hardware functionality and often reimplement features in software even if the hardware already provides them. Another inefficiency is due to copy operations between kernel- and user-space and within the kernel itself. To transmit a message the kernel has to copy the data from or to user space. The copying between protected address space boundaries often adds more latency than the physical transmission of a message. In addition, the kernel copies the data several times from one buffer to another while traversing layers of the protocol stack. On

the possitive side, the traditional communication path with the kernel as single point of access to the hardware ensures correct interaction with the hardware and mutal exclusion of competing processes.

For parallel computation on clusters of workstations, many of the protocols which are designed for wide area networks are too inefficient. Therefore, cluster computing must take new approaches.

The most promising technique is to move protocol processing to user-level. This technique opens up the opportunity to investigate optimized protocols for parallel processing. With user-level protocols there is no need to use the standardized interfaces between the operating system and the device driver. Thus, the reimplementation of services in software which are already provided by the hardware can be avoided.
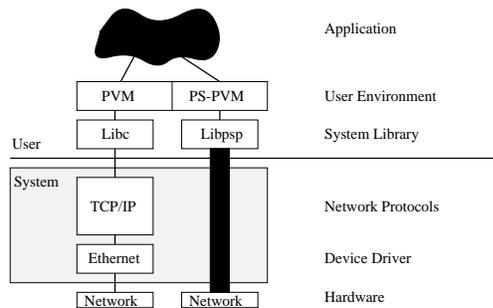

Fig. 1. User-level communication highway

User-level communication removes the kernel from the critical path of data transmission. Figure 1 shows how user-level communication shortcuts the access to the communication hardware. High-performance communication protocols are based on superior hardware features to speed up communication. Copying data between kernel- and user-space is avoided and the implementation of true zero-copy protocols is possible. These key issues minimize latency and lead to high throughput.

But user-level communication has also its drawbacks, because now the single point of access to the communication hardware, namely the kernel, is missing. Therefore many user-level communication libraries restrict the number of processes on a node to a single process. Enabling multiple processes on one node in user-level raises difficulties, but also offers a lot of benefits. Once problems, such as demultiplexing of messages and ensuring correct interaction between multiple processes are solved, the high-speed communication network can be used similar to a cluster with regular communication channels such as Unix sockets.

The goal of PULC is to provide a multi-user, multi-programming communication library for user-level communication on top of a high-speed communi-

cation hardware. The first implementation of PULC uses the ParaStation communication adapter, which is described in section 3. Section 4 presents design alternatives and the optimization techniques used. In section 5, we describe implementation of PULC on top of ParaStation. Finally, we present performance figures for two different hardware platforms (section 6).

2   Related Work

There are several approaches targeting efficient parallel computing on workstation clusters. Some of them use custom hardware which support memory mapped communication. SHRIMP[DBDF97] builds a client server computing environment on top of a virtual shared memory. Similar to PULC, SHRIMP offers standardized interfaces such as Unix sockets. Digital's Memory Channel[FG97] is proprietary to DEC Alphas and uses address space mapping to transfer data from one process to another. On top of this low level mechanism Memory Channel offers MPI and PVM. Many recent parallel machines, e.g. IBM SP2, are a collection of regular workstations connected with a high speed interconnect.

Others use commodity hardware to implement communication subsystems. OSCAR (e.g. [JR97]) implements MPI on top of SCI cards. Fast Messages [CPL+97] and Active Messages [CCHvE96] are approaches for MPP systems ported to workstation clusters. Both offer low latency protocols which can be used to build other communication libraries on top. BIP [PT97] Myricom GM [myr] implement low level interfaces to the Myrinet hardware. They are comparable with the PULC hardware abstraction layer but lack on higher protocols. Gamma [CC97] builds Active Messages on top of Fast Ethernet card and gets nearly full performance by adding a system call and building a special protocol in the Linux kernel. UNet [WBvE97] uses Fast Ethernet and ATM to build an abstraction of the network interface. Dependent on the hardware support, they use kernel or user level communication. They've even built a memory management system to enable DMA transfer to previously unpinned pages. Such a memory management is not implemented in PULC, but should be done as soon as hardware with DMA transfer is supported.

In the Berkeley NOW project [ACP95], GLUnix offers a transparent global view of a cluster. As in PULC the network of workstations can be used similar to a single parallel machine.

3   ParaStation Hardware

The first implementation of PULC uses the ParaStation high-speed communication card as communication hardware. This section gives a short overview of the ParaStation hardware. ParaStation is the reengineered MPP-network of Triton/1 [HWTP93], an MPP-system built at the University of Karlsruhe. Within a workstation cluster the ParaStation hardware is dedicated to parallel applications while the operating system continues to use standard hardware(e.g., Ethernet).

The network topology is based on a two-dimensional toroidal mesh. Table-based, self-routing packet switching transports data using virtual cut-through routing.

Buffering decouples network operation from local processing.

The size of the packet can vary from 4 to 508 bytes. Packets are delivered in order and no packets are lost. Flow control is provided at link level and the unit of flow control is one packet. These features enable the software to use a simple fragmentation/defragmentation scheme.

The communications processor used involves a routing delay of about $250ns$ per node and offers a maximum throughput of 16 Mbyte/s per link.

The ParaStation hardware resides on an interface card which plugs into the PCI-bus of the host system. Thus, it possible to use ParaStation on a wide range of machines from different vendors. A more detailed description of the hardware is given in [WBT97].

4   Design of PULC

A new communication subsystem has to fulfill several issues to be helpful for parallel computing. First, parallel computing is highly dependent on very low latency and high throughput. The performance available for the user has to be close to the hardware limits. Therefore, deep protocol stacks are poison for parallel computing.

Second, communication hardware is getting faster and more intelligent. New approaches, such as DMA transfers and communication processors on the interface cards enable high performance and flexible protocol processing. A new communication protocol has to be well-suited for these technologies.

Third, communication libraries offer different interfaces and semantics to the programmer. Not each communication library is well-suited for all users of a cluster of workstations. Therefore, a new communication subsystem has to offer different interfaces (communication libraries). It should also be extensible for new approaches in this field.

Fourth, workstation clusters are often used by several people for parallel computing. Having user level access to the hardware usually prohibits simultaneous use of one node by several processes. A new approach should support a multi process environment.

Therefore the main goal was that PULC supports fine grained parallel programming on workstation clusters while still providing the benefit of multi-process environments.

The most challenging problem in a multi-process environment is the demultiplexing of incoming messages. Generally there are three possible places where message demultiplexing can take place:

– In the operating system: The operating system either checks periodically the hardware for pending messages or it is interrupted by the hardware when a message has arrived. The operating system unpacks the message header and

stores the message data in a corresponding queue in kernel space. From the viewpoint of the kernel it doesn't matter if the message is for the currently running process or for any other process.

- In the communication processor: Each communicating process has a memory area which is accessible by the communication hardware. The communication processor checks the header and decides where the message fragment should be stored. The number of accessible memory areas is limited, however. To solve this problem the communication system can either limit the number of communicating processes or it buffers the message intermediately, where the processes can access the data (in kernel space, common message area, or a trusted process' address space).
- In the low level communication software in user space: A user process periodically checks the hardware (or gets interrupted), and receives the message. If the message is not addressed to the receiving process, the process stores the message in a message pool accessible by the destination process.

In all cases the destination process executes a receive call and gets the data from the intermediate storage and stores it into the final destination. If the final destination is known and accessible at the time of message demultiplexing, the message can be stored directly in this area. This is known as *true zero copy* [BBVvE95].

PULC divides the message demultiplexing and the message reception in two different modules. The *PULC message handler* demultiplexes incoming messages. This message handler can either run on the communication processor or it can be linked to each user process. The *PULC interface* receives the message for the process. It always runs in the address space of the communicating process.

Another challenging task is resource management. Resources are usually managed by the operating system. When moving the communication out of the kernel, this task can be accomplished by a regular user process. The resource manager has to control access to the hardware and clean up after application shutdowns. In PULC, this task is performed by the *PULC resource manager*.

Figure 2 gives an overview of the major parts of PULC.

PULC Programming Interface: This module acts as programming interface for any application. The design is not restricted to a particular interface definition such as Unix sockets. It is possible and reasonable to have several interfaces (or protocols) residing side by side, each accessible through its own API. Thus, different APIs and protocols can be implemented to support a different quality of service, ranging from standardized interfaces (i.e. TCP or UDP sockets), widely used programming environments (i.e. MPI or PVM), to specialized and proprietary APIs (ParaStation ports and a true zero copy protocol called Rawdata). All in all, the PULC interface is the programmer-visible interface to all implemented protocols.

PULC Message Handler: The message handler is responsible to handle all kind of (low level) data transfer, especially incoming and outgoing messages, and is the only part to interact directly with the hardware. It consists of a
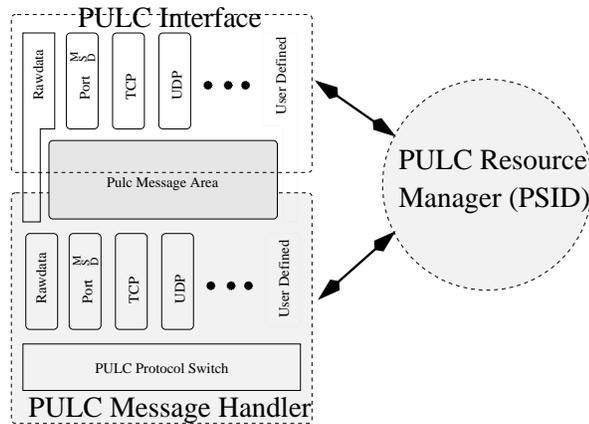
PULC Interface

Rawdata | Port MB | TCP | UDP | • • • | User Defined

Pulc Message Area

Rawdata | Port MB | TCP | UDP | • • • | User Defined

PULC Protocol Switch

PULC Message Handler

PULC Resource Manager (PSID)

Fig. 2. PULC Architecture

protocol-independent part and a specific implementation for each protocol defined within PULC. The protocol-independent part is the *protocol switch* which dispatches incoming messages and demultiplexes them to protocol specific *receive handlers*. To get high-speed communication, the protocols have to be as lean as possible. Thus, PULC protocols are not layered on top of each other; they reside side by side. Sending a message avoids any intermediate buffering. After checking the data buffer, the sender directly transfers the data to the hardware. The specific protocols inside the message handler are responsible for the coding of the protocol header information.

PULC Resource Manager: This module is implemented as a Unix daemon process (PSID) and supervises allocated resources, cleans up after application shutdowns, and controls access to common resources. Thus, it takes care of tasks usually managed by the operating system. All PSIDs are connected to each other. They exchange local information and transmit demands of local processes to the PSID of the destination node. With this cooperation, PULC offers a distributed resource management. The one system semantic of PULC is ensured by the PSIDs.

To be portable among different hardware platforms and operating systems, PULC implements all hardware and operating system specific parts in a module called hardware abstraction layer (HAL). Choosing a different interconnection network would force the adoption of the PULC message handler to the quality of service the communication hardware provides.

User level protocols often use the polling strategy to know when a new message arrives. Polling consumes CPU time while waiting for a sender. If the sender is on the same node, the sender is slowed down and it takes longer to come to the send call for which the receiver is waiting for. PULC uses different co-scheduling strategies to hand off the CPU to the sender. Therefore even local communica-

tions have acceptable latencies.

The following subsection presents an example how message transfer works and how the PULC modules interact.

4.1 The process of message transmission

The following example (see figure 3) explains how PULC processes and transmits data. For explanation we use the well-known TCP socket protocol. Two major components, the *message handler* and the *programming interface*, are clearly separated. The programming interface as communication library is part of each calling process. The message handler could either reside within the communication library (current implementation) or it could run on a communication processor within the communication adapter. Depending on the location of the message handler, the *message area* is either shared among all processes or each process has its own message area if a communication processor is capable of distributing memory to multiple areas.

The sending process invokes a send call to transmit the data to a destination which is already connected to the associated socket. The TCP interface checks the parameters and invokes the TCP message handler. The message handler determines the destination port and sends the data – possibly in multiple fragments – by invoking the send routine of the HAL. The message data is always transferred to the underlying layer by pointers without copying data.
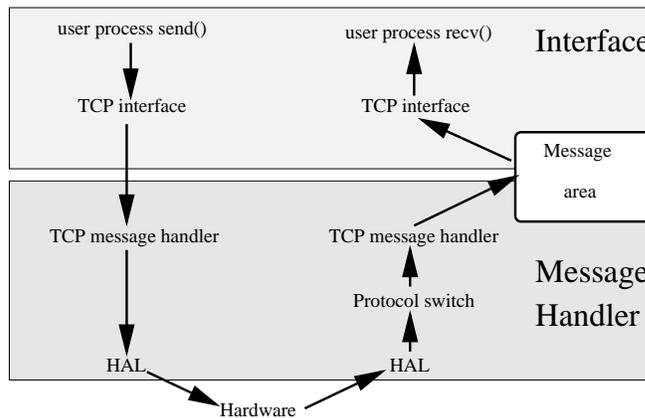

Fig. 3. TCP message transmission in PULC

At the receiving node, the receive handler checks the HAL if any message is pending. When a message arrives, the receive handler determines inside the protocol switch which protocol handles the incoming message and transfers control

to the specific receive handler. The TCP receive handler receives the body of the possibly fragmented message, determines the receiving port, and stores the fragment in the message queue associated with this port. The message queue itself is located in the message area. When the receiving process issues a receive call, the TCP interface checks the communication port of the associated socket for pending messages and delivers them to the application.

The port protocol acts in a similar way. Differences are due to new functionality which is not included in the socket standard. There are additional calls to give a view the cluster a single virtual machine and to allow more flexible receives, sends, and selects. Due to these extensions, other communication systems, such as MPI, PVM and AM can be build very easily on top of this protocol.

The rawdata protocol has some differences to allow a true zero copy protocol implementation. The sending is equivalent to the TCP protocol. On the receiver side, the rawdata receive handler directly receives in a user supplied buffer, if the location of the buffer is accessible at the time of executing the receive handler.

5 Implementation

There exists two implementations of PULC, one for Intel-PCs running Linux and the other for DEC-Alpha workstations running Digital Unix. Both of them use the ParaStation high-speed communication card as communication hardware. As described in section 3, ParaStation offers many useful services to the software protocols, but unfortunately, it has no communication processor on board. Thus, the implementation uses a commonly accessible shared memory area (see figure 4) to store messages and control information. The PULC library itself, in particular the PULC message handler, acts as the trusted base within the whole system. The library is statically linked to each application and ensures correct interaction between all parts of the system. The operating system is only invoked at system and application startup.

Operating system and hardware specific parts of the library are placed in a separate module (the HAL). Therefore only this module has to be change when porting PULC to another platform.

Since the message handler is part of each process, the message area is mapped into each communicating process. This enables the message handler to receive messages for different processes and to demultiplex them to the correct receiving port. The multi-process ability of this solution is quite expensive due to the locking of ports, as well as locking data transmission to and from the hardware.

Using a commonly accessible message area suffers from a (minimal) lack of protection. The implemented message demultiplexing implies that all communicating processes trust each other. A malfunctioning process accessing the common message area directly is able to corrupt data owned by another process and can possibly crash the system. But the risk is minimal since the address space of Alpha processors (64 bit addresses) is approximately 2^{52} times larger than the size of the message area (configuration dependent). If a wrong address is produced once a second, a corruption of data in the message area could happen
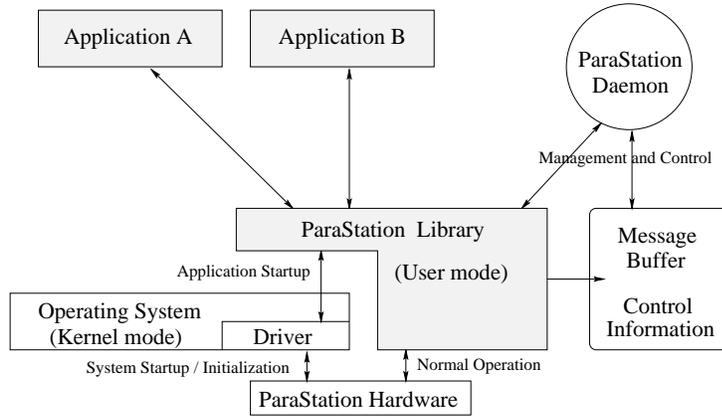
Fig. 4. ParaStation User-Level Communication

approximately every 2^{27} years. On the other hand, the trusted system is open for malicious hackers with access to the cluster, but this is a tolerable disadvantage when compared to the performance benefits gained from this policy. If this lack of protection is to harmfull, PULC can be configured to allow only a specific number of process or only a specific user access to the communication system concurrently.

6 Performance Evaluation

This section shows the efficiency of the PULC implementation. We present the performance of the different protocols and explain the results. Performance is measured on a running cluster where each node in the cluster is a fully configured workstation.

Communication subsystems can be compared by evaluating the latency and throughput of the systems. PULC offers several interfaces and runs on several hardware/operating system environments. Our test clusters consist of two Pentium PCs (166MHz) running Linux 2.0 and two Alphas 21164 (500MHz) running Digital Unix 4.0.

We compare the results with the operating system performance whenever possible. The test consists of a pairwise exchange program to measure the throughput and a ping-pong test to measure the latency. We report the round trip time divided by two. In the exchange program both processes send a message to the other and wait for the receive of the other. Therefore both processes execute always the same command. In the pingpong test, one process sends and the other receives, after receiving the message, the receiver sends the message back to the sender. Surprisingly, the slower Pentium system performs better than the Alpha system in both latency and throughput at lower layers. This is due

protocol-layer	Alpha 21164, 500 MHz				Pentium, 166 MHz			
	ParaStation		OS/Ethernet		ParaStation		OS/Ethernet	
	latency $[\mu s]$	band-width [MB/s]	latency $[\mu s]$	band-width [MB/s]	latency $[\mu s]$	band-width [MB/s]	latency $[\mu s]$	band-width [MB/s]
hardware	4.2	12.4			3.4	15.6		
rawdata	5.1	11.9			6.4	14.8		
port-M	8.9	9.5			14.6	11.8		
socket	9.0	9.6	115	1.1	13.9	11.9	308	1.0
PVM	78.0	8.7	289	1.0	158	7.8	776	0.8
PVM (port-M)	11.5	9.4			27.2	11.5		
socket (self)	3.2	318.8	390	33.0	19.0	107.0	578	30.0

to the architectural differences between the two systems. In particular Alpha's capability to combine writes to the same memory location requires additional sychronization. As the ParaStation communication interface is implemented as a fifo buffer, we had to insert memory barrier instructions (MB) after each write to the fifo. The MB instruction itself waits for all outstanding read and write operations and thus limits the performance. In addition to the write combining bottleneck, the semaphore mechanism which we use in the Alphas is not as fast as the semaphores on the Pentium. A lock operation on the Alphas takes about 1 μs whereas a Pentium provides mutual exclusion within 200 ns. The semaphore bottleneck is also visible in multi-process protocols .

The line titled *hardware* in the table above shows the performance of the hardware abstraction layer described in section 5 and reflects the maximum performance one can get using ParaStation on the stated workstation.

The additional latency of 0.9 μs on the Alpha (3 μs on the Pentium) introduced by the rawdata protocol is due to guarantee mutual exclusion and correct interaction between concurrent processes. Multiple ports are addressed by the port protocol. This multiprogramming environment adds additional 3.8 μs (8.2 μs on Pentium) to the rawdata protocol. Providing full TCP socket functionality within 9 μs opens up a wide range of fine grained parallel programs on top of sockets. As reported in [BWT96] standard programming environments, such as PVM, add a huge amount of latency to the sockets. This is not noticeable when slow operating system sockets are used. When running on top of PULC sockets 89 % (91% on a Pentium) of the latency is caused by these packages. These numbers show that these standard environments are not well suited for high speed protocols. This lead us to an improvement of PVM on top of ports. As reported, the port-M protocol already provides most of the funtionality that PVM has to implement on top of sockets, e.g. a very inefficiently implemented buffer management. Using the whole functionality of PULC, PVM only adds 2.5 μs (13.4 μs on the Pentium) to the port-M protocol latency. This shows that even with standardized interfaces, PULC offers great performance.

7 Conclusions

PULC shows extremely good performance on all protocols. Many programs benefit from the high speed of the PULC library. PULC's design offers nearly the raw performance of high-speed communication cards to the user while still providing standardized interfaces. The design goal of a multi-user/multi-programming environment at full speed was reached. PULC is also easily adapted to new hardware and brings efficient parallel processing to workstations clusters. Presented performance results compare well with parallel systems. PULC is included in the ParaStation system, which was introduced into market in 1996[1] and is currently ported to the Myrinet communication adapter.

In future, we will work on next-generation ParaStation hardware. Current issues for a new network design are fiber optic links, optimized packet switching, and flexible DMA engines to reach an application-to-application bandwidth of about 100 Mbyte/s. We will port PULC and the full ParaStation environment to other systems with PCI bus (e.g., Sun/Solaris, IBM-PowerPC/AIX, SGI/IRIX). PULC itself will be ported to other communication hardware. Additional interfaces and protocols, such as MPI, PVM, and Active Messages, are considered to be implemented as protocols inside of PULC. This would give them a performance boost over the current implementation which is implemented on top of sockets or ports.

References

[ACP95]   Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[BBVvE95] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado*, December 3-6, 1995.

[BDG⁺93]   A. Beguelin, J. Dongarra, Al Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[BWT96]   Joachim M. Blum, Thomas M. Warschko, and Walter F. Tichy. PSPVM:Implementing PVM on a high-speed Interconnect for Workstation Clusters. In *Proc. of 3rd Euro PVM Users' Group Meeting*, Munich, Germany, Oct.7-9, 1996.

[CC97]   G. Chiola and G. Ciaccio. Gamma: a low-cost network of workstations based on active messages. In *5th EUROMICRO workshop on Parallel and Distributed Processing*, 1997.

[CCHvE96] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96, Pittsburgh, PA*, November 1996.

---

[1] For further information, see `http://wwwipd.ira.uka.de/parastation`.

[CGH94]   Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message
          Passing Interface Standard. Technical report, March 94.

[CPL⁺97] Chien, Pakin, Lauria, Buchanan, Hane, Giannini, and Prusakova. High
          Performance Virtual Machines (HPVM): Clusters with Supercomputing
          APIs and Performance. In *Eighth SIAM Conference on Parallel Processing
          for Scientific Computing (PP97)*, 1997.

[DBDF97] Stefanos N. Damianakis, Angelos Bilas, Cezarz Dubnicki, and Edward W.
          Felten. Client Server Computing on Shrimp. *IEEE Micro*, pages 8–17,
          January/February 1997.

[FG97]   Marco Fillo and Richard B. Gillett. Architecture and implementation of
          memory channel 2. Technical report, Digital Equipment Coropration, 9
          1997.

[HWTP93] Christian G. Herter, Thomas M. Warschko, Walter F. Tichy, and Michael
          Philippsen. Triton/1: A massively-parallel mixed-mode computer designed
          to support high level languages. In *7th International Parallel Processing
          Symposium, Proc. of 2nd Workshop on Heterogeneous Processing*, pages
          65–70, Newport Beach, CA, April 13–16, 1993.

[JR97]   H. Jin and W. Rehm. Performance of message passing and shared mem-
          ory on sci-based smp cluster. In *Proceedings of Fifth High Performance
          Computing Symposium, Atlanta, Georgia*, April 6-10 1997.

[myr]   *The GM API.*

[PT97]   Loic Prylli and Bernard Tourancheau. New protocol design for high perfor-
          mance networking. Technical report, LIP-ENS Lyon, 69364 Lyon, France,
          1997.

[WBT97]   Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. ParaSta-
          tion: Efficient Parallel Computing by Clustering Workstations: Design and
          Evaluation. *Journal of Systems Architecture*, 1997. Elsevier Science Inc.,
          New York, NY 10010. *To appear.*

[WBvE97] Matt Welsh, Anindya Basu, and Thorsten von Eicken. ATM and Fast Eth-
          ernet Network Interfaces for user-level communication. In *roceedings of the
          Third International Symposium on High Performance Computer Architec-
          ture (HPCA), San Antonio*, 1997.

This article was processed using the LaTeX macro package with LLNCS style