Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters

Evangelos P. Markatos Manolis G.H. Katevenis*
Computer Architecture and VLSI Systems Group
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385, Science and Technology Park,
Heraklio, Crete, GR-711-10 GREECE
{markatos,katevenis}@ics.forth.gr

In the Proceedings of the HPCA-2, San Diego, Ca, Feb 1996

Abstract

Networks of workstations and high-performance microcomputers have been rarely used for running high-performance applications like multimedia, simulations, scientific and engineering applications, because, although they have significant aggregate computing power, they lack the support for efficient message-passing and shared-memory communication. In this paper we present *Telegraphos*, a distributed system that provides efficient shared-memory support on top of a workstation cluster. We focus on the network interface of Telegraphos that provides a variety of shared-memory operations like remote reads, remote writes, remote atomic operations, all launched from user level without any intervention of the operating system.

Telegraphos I, the first Telegraphos prototype has been implemented. Emphasis was put on rapid prototyping, so the technology used was conservative: FPGA's, SRAM's, and TTL buffers. Telegraphos II, is the single-chip version of the Telegraphos architecture; its switch was implemented and its network interface is being debugged.

1 Introduction

Popular contemporary computing environments are comprised of powerful workstations connected via a network which, in many cases, may have a high throughput, giving rise to systems called *workstation clusters* or Networks of Workstations (NOWs) [1]. The availability of such computing and communication power gives rise to new applications like multimedia, high performance scientific computing, real-time applications, engineering design and simulation, and so on. Up to recently, only high performance parallel processors and supercomputers were able to satisfy the computing requirements that these applications need. Fortunately, modern networks of workstations connected by Gigabit networks have the ability to run most applications that run on supercomputers, at a reasonable performance, but at a significantly lower cost.

Traditional programming environments in networks of workstations have several limitations, and cannot be used to run modern parallel applications, because they induce significant overhead. For example, most traditional environments need the intervention of the operating system to make even the simplest exchange of information between workstations. Message passing systems like PVM [11] and P4 [6] are usually implemented on top of Unix sockets which require the intervention of the operating system for each message transfer. Shared-memory systems like *Virtual Shared Memory* [9, 10, 18, 19] are based on page-fault driven page replication and invalidation to provide the shared-memory illusion: when a process wants to access non-local shared data, it page faults, the operating system replicates the page locally, marks it shared, and resumes the faulted process. When a process wants to update shared data, first it traps into the operating system and invalidates all other copies of the page, and then makes its updates. Because of the software intervention, Virtual Shared Memory has been successfully used for applications that interact rather infrequently, using weak forms of consistency.

To facilitate the development of *efficient* programming environments on distributed systems, hardware-support is being added to existing workstation clusters. In the SHRIMP project for example [4], a local page can be mapped out to another page on another workstation. Updates to the local page are snooped by the SHRIMP interface which automatically sends them to the mapped out page. Thus, passing of messages is as fast as local writes. Encore's reflective memory and PRAM [24] use a similar approach. The NOW project [1] uses fast user-level message passing via Active Messages. Although the previously described systems for workstation clusters provide efficient message passing, they have limited support for shared memory.

In this paper, we present *Telegraphos*, a distributed system that consists of network interfaces and switches for efficient support of parallel and distributed applications on a workstation cluster. We call this project Telegraphos or $T\eta\lambda\acute{\epsilon}\gamma\rho\alpha\phi o\varsigma$ from the greek words $T\eta\lambda\acute{\epsilon}$ meaning remote, and $\gamma\rho\acute{\alpha}\phi\omega$ meaning write, because the *central* operation on Telegraphos is the remote write operation. A remote

*The authors are also with the University of Crete.

write operation is triggered by a simple `store` instruction, whose argument is a memory address mapped on the physical memory of another workstation. Telegraphos also provides remote read operations, atomic operations (like `fetch_and_increment`) on remote memory locations, and a non-blocking `fetch(remote,local)` operation that copies a remote memory location into a local one. Finally, Telegraphos provides an eager-update multicast mechanism which can be used to support both multicasted message-passing, and update-based coherent shared memory.

Telegraphos provides a variety of hardware primitives which if combined with appropriate software will result in efficient support for shared-memory applications.

- On a *remote* memory access, traditional systems require the help of the operating system, which either replicates locally the remote page, and makes a local memory access, or makes the single remote access on behalf of the requesting process. To avoid this operating system overhead, Telegraphos provides the processor with the ability to make a read or write operation to a remote memory location without replicating the page locally and without any software intervention; just like shared-memory multiprocessors do [2].

- If a page is accessed by a processor frequently, it may be worthwhile to replicate the page and make all accesses to it locally. To allow informed decisions, Telegraphos provides access counters for each remotely-mapped page. Each time the processor accesses a remote page, the counter is decremented, and when it reaches zero an interrupt is sent to the processor which should probably replicate the page locally.

- Telegraphos provides a write multicast mechanism in hardware which can be used to implement one-to-many message passing operations, as well as an update-based memory coherence protocol.

To verify our architecture, we designed two prototypes of it, *Telegraphos I* and *Telegraphos II*. The first has already been built using low-integration, rapid-prototyping technology (FPGA and RAM); Telegraphos II switch has been built using ASIC technology, to achieve higher performance and integration [16, 17].

The rest of the paper is organized as follows: Section 2 presents the network interface of Telegraphos, and its support for shared memory. Special attention is given in the implementation of atomic operations and of the multicast mechanism. Section 3 presents the implementation status of Telegraphos and some preliminary performance results. Section 4 describes related projects, and finally, section 5 summarizes and concludes the paper.

2 Shared Memory Support

2.1 Traditional Systems and their Problems

Efficient support for shared memory is vital for the performance of parallel and distributed applications that run on networks of workstations *. Traditional computing environments on networks of workstations either do not support shared-memory, or provide software-supported shared memory like Virtual Shared Memory (VSM) and interrupt-driven remote memory accesses.

To avoid overhead sources related to software implementations of shared-memory primitives, we have designed, and implemented, *Telegraphos*, an architecture inspired by *shared-memory* multiprocessors, which avoids their high-cost components. Telegraphos provides load and store operations to pages that reside in memories of remote workstations. Thus, any workstation in a Telegraphos distributed system can directly access the memory of any other workstation in the same distributed system, provided that it has the right to. To enforce protection, the operating system *maps* remote pages to the page tables of those processes that have the right to access the specific remote pages.

Telegraphos I (shown in figure 1) consists of *network interface* boards that plug into the TurboChannel I/O bus of DEC Alpha workstations and *switch* boards that are connected by ribbon cables to each other and to network interfaces to form a high-speed network. The Telegraphos switches provide back-pressured flow control, deterministic routing, in-order delivery of packets, and deadlock freedom. More information about the switch architecture can be found in [16, 17].

Plugging the network interface on the I/O bus instead of the memory bus creates several challenging problems but makes the system more affordable for consumers. Almost all workstations and microcomputers provide extra slots in their I/O bus, while significantly fewer and rather expensive workstations and microcomputers provide available slots in their memory bus. Plugging the network interface of Telegraphos into the memory bus could potentially make the system more efficient, but would also severely limit its market and increase its cost.

2.2 The Network Interface

The network interface or Host Interface Board (HIB) of Telegraphos is responsible for the implementation of the following shared-memory operations:

- Efficient, non-blocking remote *write* operations that write the contents of a register into a remote memory location.

- Blocking remote *read* operations, that read the contents of a remote memory location into a local register.

- Because remote reads are blocking, a form of remote *copy* or *prefetch* operation which is equivalent to a non-blocking memory read is provided.

- Remote *atomic* operations are also implemented to allow simple and efficient synchronization.

- Because page-level replication and invalidation decisions are costly, Telegraphos helps the systems software in these decisions by providing *page access*

*The notion of workstation as used in this paper is broad enough to include high-end PCs as well.

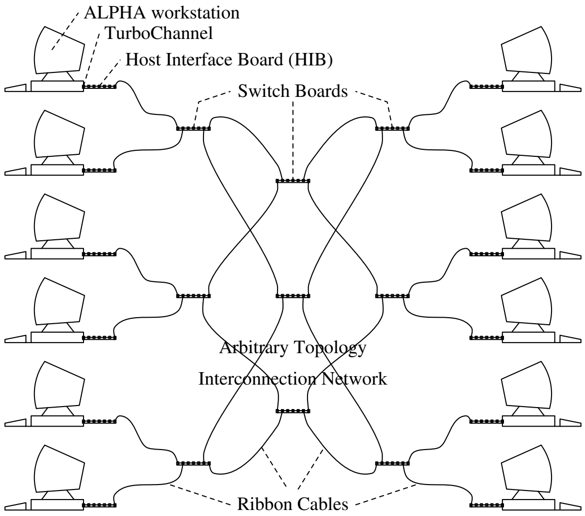

Figure 1: Example of Telegraphos I prototype configuration.

counters and *alarms*. In this way, it helps the operating system estimate the reference patterns to each page and know when the number of accesses to a page exceeds a threshold.

- To facilitate the completion detection of remote accesses, special counters of outstanding remote operations are also provided.
- Finally, Telegraphos provides *eager updating*, or *multicasting* as an efficient method to support both message passing (to multiple destinations) and coherent shared memory.

2.2.1 Remote Write - Remote Read

All remote accesses are performed via the TurboChannel of the DEC Alpha workstations on which Telegraphos HIB is plugged into. To make remote accesses visible to the HIB, remote addresses are mapped into physical addresses that correspond to the TurboChannel address space. The highest order bits of each physical address denote the node identification on which the physical memory location resides. When the HIB sees a read or write operation to a remote node, it sends a network packet with the read or write request to the HIB of the remote node. Read requests stall the processor until the data arrive from the remote node. Write requests do not stall the processor and release the TurboChannel as soon as the write request is latched by the HIB. Thus, remote writes are the most efficient operations on Telegraphos.

Mapping virtual to physical addresses is done by the operating system as follows:

- Shared data that physically reside on some remote workstation are mapped into physical addresses of the I/O bus of the workstation. Thus, when a processor wants to access shared data, its request is sent to the I/O bus. The HIB which resides in the I/O bus latches the request and services it.
- Shared data that physically reside in the local workstation are mapped in two different ways in our two prototypes:
  - Telegraphos I uses memory modules on the HIB to store shared data that reside locally.
  - Telegraphos II uses a portion of the workstation's main memory to store shared data that reside locally.

  Although the first approach results in better control over all Telegraphos operations, the second results in cacheability and faster access to shared data, better utilization of main memory, and probably better overall performance.
- Data which are *not shared* are mapped into physical addresses which correspond to the main memory of the workstation. Thus, when a processor accesses non-shared data, its access is routed to the cache (or the main memory) via the memory bus as usual. Telegraphos does not interfere with these accesses at all.

2.2.2 Remote Copy

The remote copy operation is similar to a non-blocking memory-to-memory read operation. It copies the contents of a remote memory location A_{from} to a local memory location A_{to}. The remote copy operation is launched from user level (see section 2.2.4 below); it returns control to the processor without waiting for the completion of the operation.

2.2.3 Remote Atomic Operations

To provide efficient synchronization of parallel applications, Telegraphos implements the `fetch-and-store`, `fetch-and-inc`, and `compare-and-swap` remote atomic operations.

2.2.4 Launching of Special Operations

Although remote read and write operations are launched as single (load or store) instructions, atomic operations as well as remote copy operations† need a sequence of instructions. These instructions must communicate to the HIB the following information:

- The *physical* address of the synchronization variable where the atomic operations is to be performed (two physical addresses, one for source and one for destination, for `remote_copy` operations).
- The datum of the special operation

The communication of the information to the HIB is done with a series of uncached write operations, followed by a read operation that returns the result of the atomic operation. Two are the main difficulties in doing such a sequence of operations:

†We will call both atomic operations and remote copy operations as *special* operations.

- The user should not be allowed to communicate *physical* addresses to the HIB without any protection, because a malicious or ignorant user may communicate *physical* addresses to the HIB to which he/she has no access rights.
- The sequence of write and read operations that pass the desirable information to the HIB should execute atomically, like a transaction. If only half of the sequence is executed, this would probably leave the hardware in an intermediate state, and would return erroneous results to software. Thus, the sequence of instructions that execute the special operation, should either not be interrupted, or if interrupted, resumed appropriately,

The two Telegraphos prototypes follow different approaches in dealing with these problems:

Launching Special Operations in Telegraphos I: To communicate remote physical addresses to the HIB the processor performs store operations directly to those address. To make sure that the HIB does not perform the store operation issued by the processor, the HIB is first put in a *special mode*, by writing into a special HIB address. When the HIB is in special mode, it does not perform the remote read/write operations requested by its local processor, but instead interprets them as argument passing commands. Protection checking is done at the same time: if the user has no right to access an address, the TLB will catch it and a page fault will be generated. If the user has permission to write to these addresses, the TLB will make the virtual to physical translation without generating a page-fault, the store request will be send to the TURBOchannel where the HIB will latch the physical address and use it as an argument to the special operation.

To solve the second problem (no interruption) we write the sequence of writes followed by the read in *PAL code* [25], a special mode of execution provided by the Alpha processor. A sequence of PAL code instructions is guaranteed to be executed uninterrupted. Because only the super-user has the right to install PAL code, and only at boot-time, the naive or malicious user can not tamper with the HIB and compromise the security of the system. If the process attempts to access an invalid memory location inside the PAL code, a page fault will be generated, the process will (probably) be terminated and the HIB will be restored into a clean state ‡.

Launching of Special Operations in Telegraphos II To overcome the use of PAL code, and the need for a *special mode*, we use the notions of *Telegraphos contexts* and *shadow addressing* [13]: A Telegraphos context is just a set of registers that hold the arguments of the special operations. These contexts are mapped in the virtual address space of applications, so that an application can write directly to them; an application that attempts to write to a Telegraphos context it is not allowed to, will immediately take a page fault.

‡ Some operating systems (like MACH) do not allow for page faults within PAL code. However, other operating systems (like the DEC OSF/1) generate a normal page fault when an invalid address is accessed within PAL code.

Applications that want to launch a special operation write the arguments (using a sequence of uncached writes) in their Telegraphos contexts and complete the special operation with an access to a special HIB register. If, however, an application needs to pass a *physical* address as an argument to a special operation, this is done using the notion of shadow addressing [13]. For each virtual address that maps into a physical address, we introduce a shadow virtual address that maps into a shadow physical address. An address differs from its shadow only in the highest bit. When a user application wants to pass a physical address to the Telegraphos HIB to be used as an argument to a special operation, it issues a store operation to its corresponding shadow virtual address. Telegraphos latches this store operation, gets the physical address, strips the highest order bit, and uses the remaining address as an argument to a special operation.

The argument of the store instruction contains the identification of the Telegraphos context where the physical address is to be placed, along with a key that verifies that the process issuing the store instruction is allowed to use this Telegraphos context. This combination of Telegraphos contexts, keys, and shadow addressing, albeit a little complicated, it manages to translate a virtual address to its corresponding physical one, and pass it to the network interface in a secure way, all in one store instruction issued from user-level.

If an application gets interrupted while launching a special operation, the Telegraphos contexts preserve their contents, so that the special operation will be launched when the application is resumed.

2.2.5 Related Network Interfaces

Launching of operations that need more than one instruction is a problem faced by all systems that need to provide operations that are not included in the processors instruction set.

The simplest way to launch an atomic operation is to invoke the operating system, which checks the validity of the addresses, passes the arguments to the HIB and returns the result, all uninterrupted. The obvious drawback of this approach is that special operations pay the overhead of an operating system trap, plus the page table lookup.

In SHRIMP [4] special operations are launched using the notion of *virtual memory mapped commands*. For each virtual memory page mapped to a shared physical page, there exists another virtual page (called command page) which is mapped to physical address space but not to physical memory. Accesses to a command page are not executed as regular load or store operations, but as special operations to pages that the command pages correspond to. For example, send operations based on DMA transfers are launched using accesses to the command pages that correspond to the pages to be transferred.

The FLASH [13] multiprocessor uses a programmable network interface on which a rich variety of special (programmable) operations may be supported. Communication of addresses and other information is done by using a sequence of uncached writes followed by a read to the network interface. To communicate physical addresses, FLASH uses an approach similar to virtual memory mapped commands. Because the multi-instruction sequences that

FLASH uses may be interrupted at any time, FLASH uses one operation record for each process. To maintain information across context switches and ensure authenticity, the FLASH operating system saves and restores a PID (process id) register on the network interface on every context switch. Thus, all accesses to the shadow address space place the physical address they communicate into the context that corresponds to the process indicated by the PID register at the network interface. Unfortunately, saving and restoring even a single PID during context switches, involves modification of the interrupt handler which implies that a significant part of the operating system (along with the appropriate licenses) has to be distributed along with the architecture. Although this maybe a reasonable approach for a new multiprocessor like FLASH, it is out the question for a network interface like Telegraphos, because most of the potential Telegraphos users just want a device driver to install in their systems that use Telegraphos, and may not be willing (or able) to download a new operating system into their computers. To overcome these problems, Telegraphos uses a key which provides authenticity of the process that requests the special operation. Store operations to shadow address space place the physical address they communicate to the context indicated by highest bits of the argument of the store operation. The lowest bits of the argument of the store operation constitute a key, that provides security and authenticity. Only processes that know the key that corresponds to a specific context can write physical addresses into that context.

2.2.6 Page Access Counters

The HIB maintains two counters for each remote sharable page: one that counts read operations and one that counts write operations. When the processor accesses the page remotely, the corresponding counter is decremented (unless the counter is zero). When the counter is decremented from one to zero, an interrupt is sent to the operating system. By setting the counters to very large values and periodically reading them, the system can monitor the page access, find hot-spots, display statistics, and provide useful information for profiling, performance monitoring and visualization tools. By setting the counters to small values, the operating system can implement alarm-based replication: when the number of accesses exceeds a predetermined value, the operating system is notified in order to make a replication decision [5]. Our simulation studies suggest that page access counters improve the performance of distributed shared memory applications [22], and of remote memory paging systems [21].

2.2.7 Eager Updating - Multicasting

Several parallel applications have a producer/consumer style of communication where one process computes some data, which are subsequently used by one or more other processes. To reduce the read latency of the consumer processors it is convenient to send to them the data that they will use as early as possible. To facilitate this style of communication, Telegraphos provides an *eager update - multicast* mechanism: Each local page can be mapped out to one or more remote pages. Every update made by the processor to the local page is transparently sent to all remote pages, much like remote write operations. This mechanism can be used both in message passing and in shared-memory programming paradigms.

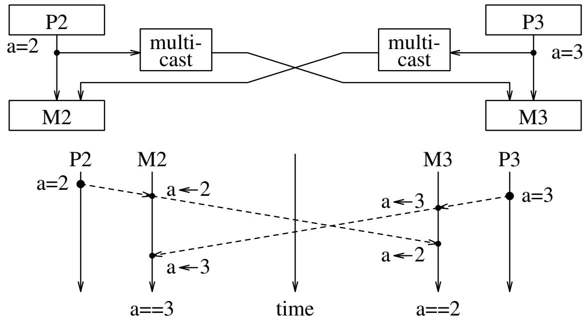

Figure 2: Inconsistency caused by multicasting in the lack of ownership.

2.3 Supporting Update-Based Coherent Memory

When multiple processors update simultaneously their own copy of the same page and multicast their updates to the other processors, the pages may end up with different values. Figure 2 illustrates how this can happen.

2.3.1 Updates through the Owner of the Page

The inconsistency just described is the result of the fact that there is no particular order in which the updates are performed [§]. Thus, updates are performed in different order in various nodes, resulting in different final values for the "copies" of the same page. To alleviate this inconsistency, we assume that for each page there is a single node which is the "owner" of the page. All updates to the copies of a page are initiated from the owner of the page, which defines the order in which all updates will be performed. This also assumes a network that delivers packets in-order from a certain source to a certain destination.

When a processor writes into a page it does not own, the write operation must be forwarded to the owner of the page. The owner must then multicast the update to all copies of the page (these multicast operations are called *reflected writes*). At the owner processor, multiple writes to the same word arrive in some particular order; by definition, this is the order in which all copies should see these writes. The owner is responsible for multicasting all packets for the same update at the same time. If a second update for the same word arrives in between, all of the new packets should be multicasted *after* all of the previous packets are sent. Since the Telegraphos network delivers packets in-order, we are assured that all copies of the page will see the two updates in the same order, and we will be left with consistent copies.

[§] In a bus-based shared-memory multiprocessor, no such inconsistency can be found, since all updates are *serialized* by the bus, and thus *all* caches see the same order of updates to a given address.

Besides maintaining consistency in the presence of multiple writers, this scheme also implies that only the owner of a page needs to hold and maintain the full list of all processors that have copies of the page. This significantly reduces the OS overhead when pages are copied, and also economizes space in the Telegraphos directories. However, not performing all write operations immediately on the local memory, but rather sending them to the owner first, introduces new problems, as discussed below.

2.3.2 Writes to Locally-Present but Remotely-Owned Pages

The existence of an owner for each page enforces all updates to be performed in the same order to all copies of the page, but it introduces a new problem:

> Suppose that a processor P has a local copy of a shared variable ***M***, but it is not the owner of the page in which M resides. Suppose that the initial value of the variable was 0. The processor executes the assignment statement $M = 1$, which sends the new value to the owner, and then P immediately reads ***M*** before the owner sends the update $M = 1$ back. If the processor P reads $M = 0$, it will be an error: The processor reads something different from what it just wrote.

One solution (with non-trivial performance cost) is to stall the read operation issued by P until the update $M = 1$ comes back from the owner, but this would make several shared-memory local read operations quite slow. A better option is to let the processor read the new value that it has just written into M, by *both* immediately performing all local writes, *and* also sending them to the owner, which will multicast them to all copies (including our own). Unfortunately, this solution introduces a new problem. Consider the following scenario:

- Processor P writes $M = 2$
- P writes $M = 3$, overwriting the first value
- the reflected value 2 returns from the owner and is written into P's memory
- P reads M expects 3 but gets 2 (error)
- the reflected value 3 returns from the owner and is written into P's memory.

In the above example, processor P writes $M = 3$, reads M, expecting to find the value of 3, but instead, it gets the value of 2, which is clearly an error.

2.3.3 Counter-based Coherent Memory

We have devised a novel solution that addresses all previously stated problems and makes sure that each processor sees a consistent view of shared memory. The intuition is as follows: During the time interval in which a processor has written a value to a shared-memory variable, but has *not* received the respective reflected write from the owner, it can just ignore *all* writes to this variable that come from the network. The reason why processor P can safely ignore all such writes is the following: Suppose that processor P has written a value v to a shared-memory location M. Suppose also, that P has sent the update to the owner O of the page, but has not yet received the multicast of its own update. Thus, any update that P receives from the network must have reached the owner O before v (otherwise the multicast for v would have arrived to P). Thus, all updates that arrive from the network are "older" than v, and there is no use in performing these updates onto variable M.

To implement the above solution, in a first, simple design, we need the following hardware: Each node P keeps, along with each memory word, N (≥ 1) extra bits, which are used to count the number of "pending writes," i.e. writes performed by the processor P whose respective multicasts (to be sent by the owner) have not yet been received by P. The counter should have enough bits to measure the maximum number of "pending writes" a processor is allowed. The details of the protocol are as follows:

1. When a processor executes a store to its local copy of a shared-memory page it does not own, it (i) updates its local copy of the page, (ii) increments the counter by one, and (iii) sends the new value to the owner of the page for multicasting.
2. When a node P receives a write from the owner of page, that is the result of one of P's own writes, P ignores the write and decrements the counter.
3. When a node receives any other write, for a memory location whose counter is non-zero, it ignores the write, without modifying the counter.
4. When a processor issues a read to a shared-memory page, the read proceeds normally.

Rules 2 and 3 make sure that each node sees a subset of the values that the owner sees, and sees them in the proper order. Thus, any value written in a memory location is always valid, and therefore, it is always safe to read it.

To implement the above protocol, we should provide a counter for each memory location. The counter should be large enough to hold the maximum number of outstanding write operations to any single memory location a processor may have. Fortunately, we do not need to keep all these counters around at any time. Keeping only a small subset of them is enough as we will show in the next section.

The run-time overhead associated with the above protocol is:

- Shared-memory read operations do not have any additional overhead. They proceed normally, reading the shared-memory and ignoring the values of the counters.
- Shared-memory write operations to pages that have remote copies incur the overhead of incrementing the counter associated with the memory location they modify. This consists of two memory accesses (one to read the counter and one to write it) and one increment operation.
- Reflected write operations arriving at the node that issued the write operation in the first place, incur the overhead of two memory accesses (one to read the

counter and one to write it), and one decrement operation.

Finally, the mentioned overhead is only paid for those operations that result in a network packet, hence their rate is bounded by the network throughput.

2.3.4 Improving the performance of counters

If the system reserved one counter for each memory location, it would spend a large percentage of memory to store counters. Fortunately, there is a small number of counters that the protocol may need at any time: only the non-zero counters are needed, which correspond to the outstanding writes that a processor may have at any time. Thus, we can use a small fast cache to hold the values of these counters:

- On a counter increment/decrement operation the counter is read from the cache, it is incremented/decremented and the new value is written back.
- If after a decrement operation, the counter reaches zero, it is not written back, and its place in the cache is marked free.
- When a counter is accessed for the first time, a new entry in the cache is allocated for it. If there is no free entry in the cache, the processor is stalled. Sooner or later, a cache entry is bound to become free, because all reflected writes from the owner are bound to arrive eventually. When an entry in the cache becomes free, the new counter is allocated there, and the processor continues its execution.

This cache can be organized as a content addressable memory. Its size can be relatively small. We expect that a cache that holds 16-32 entries will be have enough space to hold all outstanding counters for most applications.

Implementation of this cache of counters is under consideration for future versions of Telegraphos. In our first prototype, Telegraphos I, we have not implemented this cache, because we wanted to reduce the hardware complexity and the design time. Parallel applications that have at least one synchronization operation between two concurrent writes will run on top of Telegraphos I without a problem. Unfortunately, applications that have chaotic accesses may not run correctly, as their concurrent writes are not protected by synchronization. To make these applications run correctly, without our proposed cache of counters, we would have to protect their chaotic accesses with synchronization operations, which would decrease their performance.

2.3.5 Memory Consistency

Regardless of replication and the coherence protocol that may be used, all systems (like Telegraphos) that achieve fast write operations by acknowledging them immediately, may suffer from memory inconsistencies. Suppose for example that no replication is supported, and that variable `flag` resides on one processor, while variable `data` resides on another. Suppose also that processors A and B communicate with each other in the following producer/consumer style:

```
Processor A    Processor B
write(data)
write(flag)
               while(flag != OK)
                 /* spin */;
               read(data)
```

Although the `write(flag)` operations starts *after* the `write(data)` operation, it is possible that the `flag` variable is written *before* the `data` variable is written, because the communication path to the processor containing variable `flag` may be faster. Thus, processor B may read the new value of the flag and then read the *old* value of the data, effectively reading stale data. To remedy the situation, Telegraphos provides a `FENCE` or `MEMORY_BARRIER` operation. When a processor issues a `MEMORY_BARRIER` operation it is stalled until *all* its outstanding write operations have been completed. [¶] The `MEMORY_BARRIER` operation is embedded inside all implementations of synchronization operations (e.g. locks, barriers), in order to make sure that all outstanding memory accesses complete before the synchronization operation. Our example is now written as:

```
Processor A          Processor B
write(data)
UNLOCK(flag)
                     LOCK(flag)
                     read(data)
```

The `write(flag)` operation is now substituted by the UNLOCK(flag) operation which also contains a `FENCE` operation This approach makes synchronization more expensive, but keeps the cost of remote write operations low.

2.3.6 Update vs. Invalidate Coherent Memory

Although the multicast mechanism provided by Telegraphos can decrease the read latency of applications that use a producer-consumer style of communication, it may not be appropriate for applications that have different communication patterns, which may prefer an invalidate-based memory coherence protocol, or may even prohibit page replication all together, and thus eliminate the need for memory coherence. Telegraphos leaves such decisions entirely to software, and only provides mechanisms (page access counters, multicasting) that will help the system software in making correct decisions. If the software decides that the application has a producer/consumer style of communication, Telegraphos provides an efficient hardware mechanism to support it. Thus, instead of *forcing* the software to follow a particular coherence protocol, Telegraphos provides a variety of mechanisms that support, to a different extent, various coherence approaches.

2.4 Related Work in Update-Based Coherency

Update-based coherence protocols are not widespread because they are difficult to implement in large scale systems.

[¶] In the current version of Telegraphos there can be no more than one outstanding read operations.

A notable exception is the Galactica Net [15] system which implements an update-based memory coherence protocol. The protocol links all processors that share a page into a sharing ring. If two processors update the same memory location at about the same time, they will eventually notice it, because both updates will traverse the ring, and they will eventually reach both updating processors. Then, the lowest priority processor will back off. This as described in [15] guarantees that in the case where two or more processors attempt to write the same memory location at about the same time, the *final* value of the memory location that all processors see is the same.

Suppose for example, that one processor writes the value “1" to a variable, while at the same time another processor writes the value “2" to the same variable. Then under the Galactica protocol, it is possible that a third processor sees the sequence “1,2,1" which is a sequence that is not a valid program sequence under any memory consistency model. The protocol that we describe in this paper avoids this inconsistency. It makes sure that both processors read “1", or “2", or “1,2", or “2,1" which are all valid sequences, but no processor ever reads “1,2,1".

3 Telegraphos I and II Implementation

3.1 Status

Work on the Telegraphos architecture started in mid-1993. At the moment of this writing (November 1995), two prototypes of the architecture are in different stages of the implementation process. Telegraphos I has been implemented. Several operations work, and the rest are being cleaned out of the last bugs. (see figure 3).

Table 1 lists the approximate gate-count equivalent of the random logic in the various blocks of the Telegraphos I HIB; memory sizes are also shown. As seen, the portion of the network interface that is necessary for supporting shared memory is very small: 2700 gates and a few kilobits of memory. Most of it is responsible for atomic operations, while the rest is responsible for page access counters and multicasting. Telegraphos I also uses a few megabits of directory SRAM, which will usually have to reside off-chip. If the ownership-counter-based protocol is implemented in future versions of Telegraphos, the directory size will be significantly reduced.

3.2 Performance Measurements

Although Telegraphos I is still being debugged, it is stable enough to run simple experiments that measure the performance of its basic operations: remote read, and remote write.

Our experimental hardware consists of two DEC 3000 model 300 workstations connected with the Telegraphos Network. We started one application on one workstation that makes remote memory accesses to the other workstation's HIB. Remote read and write access are issued using ordinary load and store operations. After starting the application, we measured the latency of remote read and write

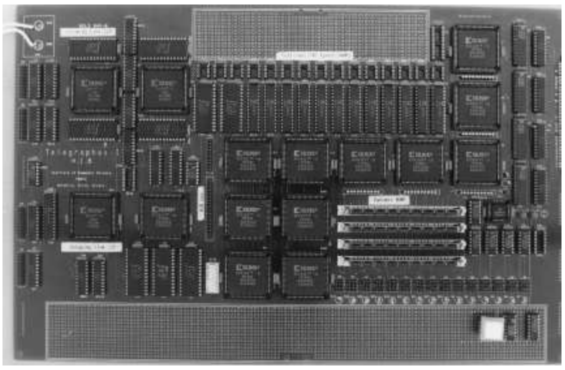

Figure 3: **Photograph of Telegraphos I Network Interface**

operations by performing 10000 operations. Our measured results are:

Operation	Elapsed Time (μsec)
Remote Read	7.2
Remote Write	0.70

We see that remote write operations are very efficient: they take less than a microsecond. The reason is that Telegraphos acknowledges a write operation as soon as its is written onto the local HIB. Thus, applications that want to send small messages can do that very efficiently. Short batches of write operations execute even faster. For example, a stream of 100 remote write operations takes less than 50 μsec, thus each of the remote write operations takes less than 0.5 μsec. The reason is that long batches of write operations are eventually performed at the network transfer rate, while short batches of write operations may take advantage of Telegraphos queueing. However, the net result is that the programmer sees that a remote write operation takes less than 0.5 μsec.

Remote read operations are less efficient: they take a few microseconds, because they need to talk to the remote HIB, read the result, communicate it to the local HIB, to the TURBOchannel, and eventually to the processor which remains blocked throughout the entire operation.

4 Related Work in Workstation Clusters

Although networks of workstations may have an (aggregate) computing power comparable to that of supercomputers (while costing significantly less), they have rarely been used to support high-performance computing, because communication on them has traditionally been very expensive. There have been several projects to provide efficient communication primitives in networks of workstations via a combination of hardware and software: PRAM [24], MERLIN [20], Galactica Net [14], Memory Channel [12], Hamlyn [7], NOW [1], and SHRIMP [4] provide efficient message passing on networks of workstations based on memory-mapped interfaces. Their shared-memory support, though, is limited because several of them do not

Block	Logic (gates)	SRAM (Kbits)	Notes:
Central control	1000	0.5	
Turbochannel interface	550		300 gates + 64 bits of registers
Incoming link intf.	1000	2.	2+2 Kb of synchr. (2-port) FIFO's
Outgoing link intf.	750	2.	
Subtotal message related	3300	4.5	
Atomic operations	1500		
Multicast (eager sharing)	400	512	16 K multicast list entries x 32 bits
Page Access Counters	800	2048	64 K pages x (16+16) bits
Multiproc. Mem. (MPM)			16 MBytes = 128 Mbits of DRAM
Subtotal shared mem. rel.	2700	2500	

Table 1: Gate Count for Telegraphos I HIB

provide atomic (and special) operations like Telegraphos does. Most of them do not even provide read operations to remote memory modules. Remote read operations should be done by replicating the page locally, making a local read, and then, either discarding the local copy, or keeping it coherent, paying the cost of memory coherence.

There have also been several implementations of software-only approaches that provide the shared-memory abstraction on a workstation cluster [3, 8, 9, 10, 18, 23]. Telegraphos builds on top of these approaches by providing hardware mechanisms (remote read/write, page access counters, multicasting), which can significantly help the system software in providing an efficient shared-memory system.

Thus, Telegraphos is an integrated hardware and software solution for shared-memory support in a workstation cluster. It provides the same functionality and efficiency as shared-memory multiprocessors, but at a significantly lower cost.

5 Summary

In this paper we described *Telegraphos*, a system for efficient support of shared-memory applications on top of a workstation cluster. Telegraphos provides a variety of shared-memory operations like remote read, remote write, prefetch, and eager-updating. No software is involved in performing all shared-memory operations, apart from the initialization phase that maps the shared pages, so that each processor can only access memory that is allowed to.

In our first prototype implementation, the Telegraphos I network interface (HIB) board plugs into the TurboChannel I/O bus of DEC Alpha 3000 model 300 (Pelican) workstations. All workstations are connected with ribbon cables and switches. Telegraphos I has been implemented and is being cleaned out of the last bugs. Telegraphos II, a single chip prototype, is being designed.

Telegraphos provides hardware support for the necessary shared-memory operations (like remote read/write and coherence messages), while leaving complicated coherence decisions to software and to users that are willing to pay the cost of coherence if they are going to benefit from it. Telegraphos provides comparable efficiency to that of shared-memory multiprocessors because it provides in hardware several of the primitives first implemented in shared-memory multiprocessors, but has significantly lower cost because (i) it does not implement hardware cache-coherence, and (ii) it uses existing workstations for computing power and main memory support.

Telegraphos is a system inspired by large-scale multiprocessors, but avoids their high costs. Thus, it can be used to develop affordable parallel processing systems in the form of workstation clusters.

Acknowledgments

This work was developed in the ESPRIT/HPCN project "SHIPS", and will be used by the IT project "ARCHES" (ESPIRIT 20693), funded by the European Union. We deeply appreciate this financial support, without which Telegraphos would have not existed.

Telegraphos is a collective effort, with many contributors. The authors wish to acknowledge in particular S. Kapidakis, P. Constanta-Fanouraki, and Ch. Nikolaou for valuable contributions to the Telegraphos design; G. Kalokerinos, M. Stratakis, Ch. Xanthaki, A. Dollas, and G. Papadourakis for implementing the rest of the Telegraphos I system prototype; G. Kornaros and N. Houstis for porting the Telegraphos I switch design to the Telegraphos II ASIC technology; and Ch. Kozirakis and M. Grammatikakis for contributing to the switch simulation. Also, the comments of D. Serpanos, M. Grammatikakis, and the anonymous reviewers were valuable. We thank all of them.

References

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[2] BBN Advanced Computers Inc. *Inside the TC2000^{TM} Computer*. Cambridge, Massachusetts, February 1990.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the COMPCON 93*, 1993.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the Twenty-First Int. Symposium on Computer Architecture*, pages 142–153, Chicago, IL, April 1994.

[5] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, Santa Clara, CA, April 1991.

[6] Ralph Butler and Ewing Lusk. User's Guide to the P4 Parallel Programming System. Technical report, Argonne National Laboratory, October 1992.

[7] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: a High-performance Network Interface, with Sender-Based Memory Management. In *Proceedings of the Hot Interconnects III*, August 1995.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] G. Delp. *The Architecture and implementation of Memnet: A High-Speed Shared Memory Computer Communication Network*. PhD thesis, University of Delaware, 1988.

[10] A. Forin, J. Barrera, and R. Sanzi. The Shared Memory Server. *Proceedings of the USENIX Winter '89 Technical Conference*, pages 229–244, January 1989.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 User's Guide and Reference Manual. Technical report, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.

[12] R. Gillet. Memory Channel. In *Proceedings of the Hot Interconnects III*, August 1995.

[13] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proc. of the 6-th International Conference on Architectural Support for Programming Languages and Operating Systems.*, pages 38–50, 1994.

[14] Andrew W. Wilson Jr., Richard P. LaRowe Jr., and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *PROC of the Thirteenth International Conference on Distributed Computing Systems*, pages 246–255, Pittsburgh, PA, May 1993.

[15] A.W. Wilson Jr., R.P. LaRowe Jr., R.J. Ionta, R.P. Valentino, B. Hu, P.R. Breton, and P. Lau. Update Propagation in the Galactica Net Distributed Shared Memory Architecture. Technical report, Center for High Performance Computing, Worchester Polytechnic Institute, 1993.

[16] M. Katevenis, P. Vatsolaki, and A. Efthymiou. Pipelined Memory Shared Buffer for VLSI Switches. In *Proceedings of the ACM SIGCOMM '95 Conference*, pages 39–48, August 1995. URL: file://ftp.ics.forth.gr/tech-reports/1995/1995.SIGCOMM95.PipeMemoryShBuf.ps.gz.

[17] M. Katevenis, P. Vatsolaki, A. Efthymiou, and M. Stratakis. VC-level Flow Control and Centralized Buffering. In *Proceedings of the Hot Interconnects III Symposium*, August 1995. URL: file://ftp.ics.forth.gr/tech-reports/1995/1995.HOTI.VCflowCtrlTeleSwitch.ps.gz.

[18] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, 1994.

[19] K. Li and P. Hudak. Memeory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[20] C. Maples. A High-Performance Memory-Based Interconnection System for Multicomputer Environments. In *Proceedings of the Supercomputing Conference*, pages 295–304, 1992.

[21] E.P. Markatos. Using Remote Memory to avoid Disk Thrashing: A Simulation Study. In *Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96)*, February 1996.

[22] E.P. Markatos and C.E. Chronaki. Trace-Driven Simulations of Data-Alignment and Other Factors affecting Update and Invalidate Based Coherent Memory. In *Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 44–52, January 1994.

[23] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level Shared-Memory. *Proceedings of the Twenty-First ISCA*, pages 325–336, April 1994.

[24] D. Serpanos. *Scalable Shared-Memory Interconnections*. PhD thesis, Princeton University, Dept. of Computer Science, October 1990.

[25] R. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.