The Need for Asynchronous, Zero-Copy Network I/O

Problems and Possible Solutions

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

Abstract

The network interfaces provided by today's OSes severely limit the efficiency of network programs. The kernel copies the data coming in from network interface at least once internally before making the data available in the user-level buffer. This article explains the problems and introduces some possible solutions. These necessarily cover more than just the network interfaces themselves, there is a bit more support needed.

1 Introduction

Writing scalable network applications is today more challenging than ever. The problem is the antiquated (standardized) network API. The Unix socket API is flexible and remains usable but with ever higher network speeds, new technologies like interconnects, and the resulting expected scalability we reach the limits. CPUs and especially their interface to the memory subsystem are not capable of dealing with the high volume of data in the available short timeframes.

What is needed is an asynchronous interface for networking. The asynchronicity would primarily be a means to avoid unnecessary copying of data. It also would help to avoid congestion since network buffers can earlier be freed which in turn ensures that retransmits due to full network buffers are minimized.

Existing interfaces like the POSIX AIO functions fall short of providing the necessary functionality. This is not only due to the fact that pre-posting of buffers is only possible at a limited scale. A perhaps bigger problem is the expensive event handling. The event handling itself has requirements and challenges which currently cannot be worked around (like waking up too many waiters).

In the remainder of the paper we will see the different set of interfaces which are needed:

- event handling
- physical memory handling
- asynchronous network interfaces

The event handling must work with the existing `select()`/`poll()` interfaces. It also should be generic enough to be usable for other events which currently do not map to file descriptors in the moment (like message queues, futexes, etc). This way we might *finally* have one unified inner loop in the event handling of a program.

Physical memory suddenly becomes important because network devices address only physical memory. But physical memory is invisible in the Unix ABI and this is very much welcome. If this is not the case the Copy-On-Write concept used on CPUs with MMU-support would not work. The implementation of functions like `fork()` becomes harder. The proposal will center around making physical memory regions objects the kernel knows to handle.

Finally, using event and physical memory handling, it is possible to define sane interfaces for asynchronous network handling. In the following sections we will see some ideas on how this can happen.

It is not at all guaranteed that these interfaces will stand the test of time or will even be implemented. The intention of this paper is to get the ball rolling because the problems are pressing and we definitely need something along these lines. Starting only from the bottom (i.e., from the kernel implementation) has the danger of ignoring the needs of programmers and might miss the bigger picture (e.g., integration into a bigger event handling scheme, for instance).

2 The Existing Implementation

Network stacks in Unix-like OSes have more or less the same architecture today as they had 10–20 years ago. The interface the OS provides for reading and writing from and to network interfaces consists of the interfaces in Table 1.

These interfaces all work synchronously. The interfaces for reading return only when data is available or in error conditions. In non-blocking mode they can return immediately if no data is available, but this is no real asynchronous handling. The data is not transferred to userlevel in an asynchronous fashion.

receiving	sending
`read()`	`write()`
`recv()`	`send()`
`recvfrom()`	`sendto()`
`recvmsg()`	`sendmsg()`

Table 1: Network APIs

receiving	sending
`read()`[1]	`write()`[1]
`aio_read()`	`aio_write()`
`lio_listio()`	

Table 2: AIO APIs

Linux provides an asynchronous mode for terminals, sockets, pipes, and FIFOs. If a file descriptor has the `O_ASYNC` flag set calls to `read()` and `write()` immediately return and the kernel notifies the program about completion by sending a signal. This is a slightly more complicated and more restrictive version of the AIO interfaces than when using `SIGEV_SIGNAL` (see below) and therefore suffers in addition to its own limitation of those of `SIGEV_SIGNAL`.

For truly asynchronous operations on files the POSIX AIO functions from Table 2 are available. With these interfaces it is possible to submit a number of input and output requests on one or more file descriptors. Requests are filled as the data becomes available. No particular order is guaranteed but requests can have priorities associated with them and the implementation is supposed to order the requests by priority. The interfaces also have a synchronous mode which comes in handy from time to time. Interesting here is the asynchronous mode. The big problem to solve is that somehow the program has to be able to find out when the submitted requests are handled. There are three modes

[1] With the `O_ASYNC` flag set for the descriptor.

defined by POSIX:

SIGEV_SIGNAL The completion is signaled by sending a specified signal to the process. Which thread receives the signal is determined by the kernel by looking at the signal masks. This makes it next to impossible to use this mechanism (and O_ASYNC) in a library which might be linked into arbitrary code.

SIGEV_THREAD The completion is signaled by creating a thread which executes a specified function. This is quite expensive in spite of NPTL.

SIGEV_NONE No notification is sent. The program can query the state of the request using the aio_error() interface which returns EINPROGRESS in case the request has not yet been finished. This is also possible for the other two modes but it is crucial for SIGEV_NONE.

The POSIX AIO interfaces are designed for file operations. Descriptors for sockets might be used with the Linux implementation but this is not what the functions are designed for and there might be problems.

All I/O interfaces have some problems in common: the caller provides the buffer into which the received data is stored. This is a problem in most situation for even the best theoretical implementation. Network traffic arrives asynchronously, mostly beyond the control of the program. The incoming data has to be stored somewhere or it gets lost.

To avoid copying the data more than once it would therefore be necessary to have buffers usable by the user available right the moment when the data arrives. This means:

- for the read() and recv() interfaces it would be necessary that the program is making such a call just before when the data arrives. If there is no such call outstanding the kernel has to use its own buffers or (for unreliable protocols) it can discard the data.
- with aio_read() and the equivalent lio_listio() operation it is possible to pre-post a number of buffers. When the number goes down more buffers can be pre-posted. The main problem with these interfaces is what happens next. Somehow the program needs to be notified about arrival of data. The three mechanisms described above are either based on polling (SIGEV_NONE) or are far too heavy-weight. Imagine sending 1000s of signals a second corresponding to the number of incoming packages. Creating threads is even more expensive.

Another problem is that for unreliable protocols it might be more important to always receive the last arriving data. It might contain more relevant information. In this case data which arrived before should be sacrificed.

A second problem all implementations have in common is that the caller can provide arbitrary memory regions for input and output buffer to the kernel. This is in general wanted. But if the network hardware is supposed to transfer directly into the memory regions specified it is necessary for the program to use memory that is special. The network hardware uses Direct Memory Access (DMA) to write into RAM instead of passing data through the CPU. This happens at a level below the virtual address space management, DMA only uses physical addresses.

Besides possible limitations on where the RAM for the buffers is located in the physical address

space, the biggest problem is that the buffers must remain in RAM until used. Ordinarily userlevel programs do not see physical RAM; the virtual address is an abstraction and the OS might decide to remove memory pages from RAM to make room for other processes. If this would appear while an network I/O request is pending the DMA access of the network hardware would touch RAM which is now used for something else.

This means while buffers are used for DMA they must not be evicted from RAM. They must be locked. This is possible with the `mlock()` interface but this is a privileged operation. If a process would be able to lock down arbitrary amounts of memory it would impact all the other processes on the system which would be starved of resources. Recent Linux kernels allow unprivileged processes to lock down a modest amount of memory (by default eight pages or so) but this would not be enough for heavily network oriented applications.

The POSIX AIO interfaces certainly show the way for the interfaces which can solve the networking problems. But we have to solve several problems:

- make DMA-ready memory available to unprivileged applications;
- create an efficient event handling mechanism which can handle high volumes of events;
- create I/O interfaces which can use the new memory and event handling. As a bonus they should be usable for disk I/O as well.

At this point it should be mentioned that a working group of the OpenGroup, the Interconnect Software Consortium, tried to tackle this problem. The specification is available from their website at `http://www.opengroup.org/icsc/`. They arrived at the same set of three problems and proposed solutions. Their solutions are not implemented, though, and they have some problems. Most importantly, the event handling does not integrate with the file-descriptor-based event handling.

3 Memory Handling

The main requirement on the memory handling is to provide memory regions which are available at userlevel and which can be directly accessed by hardware other than the processor. Network cards and disk controllers can transfer data without the help of the CPU through DMA. DMA addresses memory based on the physical addresses. It does not matter how the physical memory is currently used. If the virtual memory system of the OS decides that a page of RAM should be used for some other purpose the devices would overwrite the new user's memory unless this is actively prevented. There is no demand-paging as for the userlevel code.

To be sure the DMA access will use the correct buffer, it is necessary to prevent swapping the destination pages out. This is achieved by using `mlock()`. Memory locking depletes the amount of RAM the system can use to keep as much of the combined virtual memory of all processes in RAM. This can severely limit the performance of the system or eventually prevent it from making any progress. Memory locking is therefore a privileged operation. This is the first problem to be solved.

The situation is made worse by the fact that locking can only be implemented on a per-page-basis. Locking one small object on a page ties down the entire page.

One possibility would be avoid locking pages in the program and have the kernel instead do the work all by itself and on-demand. That means if a network I/O request specifies a buffer the kernel could automatically make sure that the memory page(s) containing the buffer is locked. This would be the most elegant solution from the userlevel point-of-view. But it would mean significant overhead: for every operation the memory page status would have to be checked and if necessary modified. Network operations can be frequent and multiple buffers can be located on the same page. If this is known the checks performed by the kernel would be unnecessary and if they are performed the kernel must keep track how many DMA buffers are located on the page. This solution is likely to be unattractive.

It is possible to defer solving this problem, fully or in part, to the user. In the least accommodating solution, the kernel could simply require the userlevel code to use `mmap()` and `mprotect()` with a new flag to create DMA-able memory regions. Inside these memory regions the program can carve out individual buffers, thereby mitigating the problem of locking down many pages which are only partially used as buffers. This solution puts all the burden on the userlevel runtime.

It also has a major disadvantage. Pages locked using `mlock()` are locked until they are unlocked or unmapped. But for the purpose of DMA the pages need not be permanently locked. The locking is really only needed while I/O requests using DMA are being executed. For the network I/O interfaces we are talking about here the kernel always knows when such a request is pending. Therefore it is theoretically possible for the kernel to lock the pages on request. For this the pages would have to be specially marked. While no request is pending or if a network interface is used which does not provide DMA access the virtual memory subsystem of the OS can move the page around in physical memory or even swap it out.

One relatively minor change to the kernel could allow for such optimizations. If the `mmap()` call could be passed a new flag `MAP_DMA` the kernel would know what the buffer is used for. It could keep track of the users of the page and avoid locking it unless it is necessary. In an initial implementation the flag could be treated as an implicit `mlock()` call. If the flag is correctly implemented it would also be possible to specify different limits on the amount of memory which can be locked and for DMA respectively. This is no full solution to the problem of requiring privileges to lock memory, though (an application could simply have a `read()` call pending all the time).

The `MAP_DMA` flag could also help dealing with the effects of `fork()`. The POSIX specification requires that no memory locking is inherited by the child. File descriptors are inherited on the other hand. If parts of the solution for the new network interfaces uses file descriptors (as it is proposed later) we would run into a problem: the interface is usable but before the first use it would be necessary to re-lock the memory. With the `MAP_DMA` flag this could be avoided. The memory would simply automatically re-locked when it is used in the child for the first time. To help in situations where the memory is not used at all after `fork()`, for example, if an `exec` call immediately follows, all `MAP_DMA` memory is unlocked in the child.

Using this one flag alone could limit the performance of the system, though. The kernel will always have to make sure that the memory is locked when an I/O request is pending. This is overhead which could potentially be a limiting factor. The programmer oftentimes has better knowledge of the program semantics. She would know which memory regions are used for longer periods of time so that one explicit

lock might be more appropriate than implicit locking performed by the kernel.

A second problem is fragmentation. A program is usually not one homogeneous body of code. Many separate libraries are used which all could perform network I/O. With the `MAP_DMA` method proposed so far each of the libraries would have to allocate its own memory region. This use of memory might be inefficient because of the granularity of memory locking and because not all parts of the program might need the memory concurrently.

To solve the issue, the problem has to be tackled at a higher level. We need to abstract the memory handling. Providing interfaces to allocate and deallocate memory would give the implementation sufficient flexibility to solve these issues and more. The allocation interfaces could still be implemented using the `MAP_DMA` flag and the allocation functions could "simply" be userlevel interfaces and no system calls. One possible set of interfaces could look like this:

```
int dma_alloc(dma_mem_t *handlep,
  size_t size, unsigned int flags);
int dma_free(dma_mem_t handle,
  size_t size);
```

The interfaces which require DMA-able memory would be passed a value of type `dma_mem_t`. How this handle is implemented would be implementation defined and could in fact change over time. An initial, trivial implementation could even do without support for something like `MAP_DMA` and use explicit `mlock()` calls.

`epoll_wait()`	`poll()`	`select()`
`epoll_pwait()`	`ppoll()`	`pselect()`

Table 3: Notification APIs

4 Event Handling

The existing event handling mechanisms of POSIX AIO uses polling, signals, or the creation of threads. Polling is not a general solution. Signals are not only costly, they are also unreliable. Only a limited, small number of signals can be outstanding at any time. Once the limit is reached a program has to fall back on alternative mechanisms (like polling) until the situation is rectified. Also, due to the limitations imposed on code usable in signal handlers, writing programs using signal notification is awkward and error-prone. The creation of threads is even more expensive and despite the speed of NPTL has absolute no chance to scale with high numbers of events.

What is needed is a completely new mechanism for event notification. We cannot use the same mechanisms as used for synchronous operations on a descriptor for a socket or a file. If data is available and can be sent, this does not mean that an asynchronously posted request has been fulfilled.

The structure of a program designed to run on a Unix-y system requires that the event mechanism can be used with the same interfaces used today for synchronous notification (see Table 3). It would be possible to invent a completely new notification handling mechanism and map the synchronous file descriptor operation to it. But why? The existing mechanism work nicely, they scale well, and programmers are familiar with them. It also means existing code does not have to be completely rewritten.

Creating a separate channel (e.g., file descriptor) for each asynchronous I/O request is not

scalable. The number of I/O requests can be high enough to forbid the use of the `poll` and `select` interfaces. The `epoll` interfaces would also be problematic because for each request the file descriptor would have to be registered and later unregistered. This overhead is too big. Furthermore, is file descriptor has a certain cost in the kernel and therefore the number is limited.

What is therefore needed is a kind of bus used to carry the notifications for many requests. A mechanism like `netlink` would be usable. The `netlink` sockets receive broadcast traffic for all the listeners and each process has to filter out the data which it is interested in. Broadcasting makes `netlink` sockets unattractive (at best) for event handling. The possible volume of notifications might be overwhelming. The overhead for the unnecessary wake-ups could be tremendous.

If filtering is accepted as not being a viable implementation requirement we have as a requirement for the solution that each process can create multiple, independent event channel, each capable of carrying arbitrarily many notification events from multiple sources. If we would not be able to create multiple independent channels a program could not concurrently and uncoordinatedly create such channels.

Each channel could be identified by a descriptor. This would then allow the use of the notification APIs in as many places as necessary independently. At each site only the relevant events are reported which allows the event handling to be as efficient as possible.

An event is not just an impulse, it has to transmit some information. The request which caused the event has to be identified. It is usually[2] regarded best to allow the programmer add additional information. A single pointer is sufficient, it allows the programmer to refer to additional data allocated somewhere else. There is no need to allow adding an arbitrary amount of data. The event data structure can therefore be of fixed length. This simplifies the event implementation and possibly allows it to perform better. If the transmission of the event structure would be implemented using socket the `SOCK_SEQPACKET` type can be used. The structure could look like this:

```
typedef struct event_data {
  enum { event_type_aio,
         event_type_msq,
         event_type_sig } ev_type;
  union {
    aio_ctx_t *ev_aio;
    mqd_t *ev_msq;
    sigevent_t ev_sig;
  } ev_un;
  ssize_t ev_result;
  int ev_errno;
  void *ev_data;
} event_data_t;
```

This structure can be used to signal events other than AIO completion. It could be a general mechanism. For instance, there currently is no mechanism to integrate POSIX message queues into `poll()` loops. With an extension to the `sigevent` structure it could be possible to register the event channel using the `mq_notify()` interface. The kernel can be extended to send events in all kinds of situations.

One possible implementation consists of introducing a new protocol family `PF_EVENT`. An event channel could then be created with:

```
int efd = socket(PF_EVENT,
     SOCK_SEQPACKET, 0);
```

[2] See the `sigevent` structure.

```
int ev_send(int s, const void *buf, size_t len, int flags, ev_t ec,
  void *data);
int ev_sendto(int s, const void *buf, size_t len, int flags,
  const struct sockaddr *to, socklen_t tolen, ev_t ec, void *data);
int ev_sendmsg(int s, const struct msghdr *msg, int flags, ev_t ec,
  void *data);
int ev_recv(int s, void *buf, size_t len, int flags, ev_t ec,
  void *data);
int ev_recvfrom(int s, void *buf, size_t len, int flags,
  struct sockaddr *to, socklen_t tolen, ev_t ec, void *data);
int ev_recvmsg(int s, struct msghdr *msg, int flags, ev_t ec,
  void *data);
```

Figure 1: Network Interfaces with Event Channel Parameters

The returned handle could be used in `poll()` calls and be used as the handle for the event channel. There are two potential problems which need some thought:

- The kernel cannot allow the event queue to take up arbitrary amounts of memory. There has to be an upper limit on the number of events which can be queued at the same time. When this happens a special event should be generated. It might be possible to use out-of-band notification for this so that the error is recognized right away.
- The number of events on a channel can potentially be high. In this case the overhead of all the `read()`/`recv()` calls could be a limiting factor. It might be beneficial to apply some of the techniques for the network I/O discussed in the next section to this problem as well. Then it might be possible to poll for new events without the system call overhead.

To enable optimizations like possible userlevel-visible event buffers the actual interface for the event handling should be something like this:

```
ec_t ec_create(unsigned flags);
int ec_destroy(ec_t ec);
int ec_to_fd(ec_t ec);
int ec_next_event(ec_t ec,
  event_data_t *d);
```

The `ec_to_fd()` function returns a file descriptor which can be used in `poll()` or `select()` calls. An implementation might choose to make this interface basically a no-op by implementing the event channel descriptor as a file descriptor. The `ec_next_event()` function returns the next event. A call might result in a normal `read()` or `recv` call but it might also use a user-level-visible buffer to avoid the system call overhead. The events signaled by `poll()` etc can be limited to the arrival of new data. I.e., the userlevel code is responsible for clearing the buffers before waiting for the next event using `poll()`. The kernel is involved in the delivery of new data and therefore this type of event can be quite easily be generated.

Handles of type `ec_t` can be passed to the asynchronous interfaces. The kernel can then

```
int aio_send(struct aiocb *aiocbp, int flags);
int aio_sendto(struct aiocb *aiocbp, int flags,
  const struct sockaddr *to, socklen_t tolen);
int aio_sendmsg(struct aiocb *aiocbp, int flags);
int aio_recv(struct aiocb *aiocbp, int flags);
int aio_recvfrom(struct aiocb *aiocbp, int flags, struct sockaddr *to,
  socklen_t tolen);
int aio_recvmsg(struct aiocb *aiocbp, int flags);
```

Figure 2: Network Interfaces matching POSIX AIO

create appropriate events on the channel. There will be no fixed relationship between the file descriptor or socket used in the asynchronous operation and the event channel. This gives the most flexibility to the programmer.

5 I/O Interfaces

There are several possible levels of innovation and complexity which can go into the design of the asynchronous I/O interfaces. It makes sense to go through them in sequence of increasing complexity. The more complicated interfaces will likely take advantage of the same functionality the less complicated need, too. Mentioning the new interfaces here is not meant to imply that all interfaces should be provided by the implementation.

The simplest of the interfaces can extend the network interfaces with asynchronous variants which use the event handling introduced in the previous section. One possibility is to extend interfaces like `recv()` and `send()` to take additional parameters to use event channels. The result is seen in Figure 1.

Calls to these functions immediately return. Valid requests are simply queued and the notifications about the completion are sent via the event channel `ec`. The `data` parameter is the additional value passed back as part of the `event_data_t` object read from the event channel. The event notification would signal the type of operation by setting `ev_type` appropriately. Success and the amount of data received or transmitted are stored in the `ev_errno` and `ev_result` elements.

There are two objections to this approach. First, the other frequently used interfaces for sockets (`read()` and `write()`) are not handled. Although their functionality is a strict subset of `recv()` and `send()` respectively it might be a deterrent. The second argument is more severe: there is no justification to limit the event handling to network transfer. The same functionality would be "nice to have"™ for file, pipe, and FIFO I/O. Extending the `read()` and `write()` interfaces in the same way as the network I/O interfaces makes no sense, though. We already have interfaces which could be extended.

With a simple extension of the `sigevent` structure we can reuse the POSIX AIO interfaces. All that would be left to do is to define appropriate versions of the network I/O interfaces to match the existing POSIX AIO interfaces and change the `aiocb` structure slightly. The new interfaces can be seen in Figure 2. The `aiocb` structure needs to have one additional element:

```
struct aiocb {
  ...
  struct msghdr *aio_msg;
  ...
};
```

It is used in the `aio_sendmsg()` and `aio_recvmsg()` calls. The implementation can chose to reuse the memory used for the `aio_buf` element because it never gets used at the same time as `aio_msg`. The other four interfaces use `aio_buf` and `aio_nbytes` to specify the source and destination buffer respectively.

The `<signal.h>` header has to be extended to define `SIGEV_EC`. If the `sigev_notify` element of the `sigevent` structure is set to this value the completion is signal by an appropriate event available on an event channel. The channel is identified by a new element which must be added to the `sigevent` structure:

```
struct sigevent {
  ...
  ec_t sigev_ec;
  ...
};
```

The additional pointer value which is passed back to the application is also stored in the `sigevent` structure. The application has to store it in `sigev_value.sival_ptr` which is in line with all the other uses of this part of the `sigevent` structure.

Introducing these additional AIO interfaces and the `SIGEV_EC` notification mechanism would help to solve some problems.

- programs could get more efficient notification of events (at least more efficient than signals and thread creation), even for file I/O;
- network operations which require the extended functionality of the `recv` and `send` interfaces can be performed asynchronously;
- by pre-posting buffers with `aio_read()` or the `aio_recv()` and now the `aio_recv` and `aio_send` interfaces network I/O might be able to avoid intermediate buffers.

Especially the first two points are good arguments to implement these interfaces or at the very least allow the existing POSIX AIO interfaces use the event channel notification. As explained in section section 2 the memory handling of the POSIX AIO functions makes direct use by the network hardware cumbersome and slower than necessary. Additionally the system call overhead is high when many interfaces use the event channel notification. As explained network requests have to be submitted. This can potentially be solved by extending the `lio_listio()` interface to allow submit multiple requests at once. But this will not solve the problem of the resulting event notification storm. For this we need more radical changes.

6 Advanced I/O Interfaces

For the more advanced interfaces we need to integrate the DMA memory handling into the I/O interfaces. We need to consider synchronous and asynchronous interfaces. We could ignore the synchronous interfaces and require the use of `lio_listio` or an equivalent interface but this is a bit cumbersome to use.

```
int dma_assoc(int sock, dma_mem_t mem, size_t size, unsigned flags);
int dma_disassoc(int sock, dma_mem_t, size_t size);
```

Figure 3: Association of DMA-able memory to Sockets

```
int sio_reserve(dma_mem_t dma, void **memp off, size_t size);
int sio_release(dma_mem_t dma, void *mem, size_t size);
```

Figure 4: Network Buffer Memory Management

For network interfaces it is ideally the interface which controls the memory into which incoming data is written. Today this happens with buffers allocated by and under full control of the kernel. It is conceivable to allow applications to allocate buffers and assign them to a given interface. This is where `dma_alloc()` comes in. The latter possibility has some distinct advantages; mainly, it gives the program the opportunity to influence the address space layout. This can be necessary for some programs.[3]

It is usually not possible to associate each network interface with a userlevel process. The network interface is in most cases a shared resource. The usual Unix network interface rules therefore need to be followed. A userlevel process opens a socket, binds the socket to a port, and it can send and receive data. For the incoming data the header decides which port the remote party wants to target. Based on the number, the socket is selected. Therefore the association of the DMA-able buffer should be with a socket. What is needed are interfaces as can be seen in Figure 3. It probably should be possible to associate more than one DMA-able memory region with a socket. This way it is possible to dynamically react to unexpected network traffic volume by adding additional buffers.

Once the memory is associated with the socket the application cannot use it anymore as it pleases until `dma_disassoc()` is called. The kernel has to be notified if the memory is written to and the kernel needs to tell the application when data is available to be read. Otherwise the kernel might start using a DMA memory region which the program is also using, thus overwriting the data. We therefore need at least interfaces as shown in Figure 4. The `sio_reserve()` interface allows to reserve (parts of) the DMA-able buffer for writing by the application. This will usually be done in preparation of a subsequent send operation. The `dma` parameter is the value returned by a previous call to `dma_alloc()`. We use a `size` parameter because this allows the DMA-able buffer to be split into several smaller pieces. As explained in section 3 it is more efficient to allocate larger blocks of DMA-able memory instead of many smaller ones because memory locking only works with page granularity. The implementation is responsible for not using the same part of the buffer more than once at the same time. A pointer to the available memory is returned in the variable pointed to by `memp`.

When reading from the network the situation is reversed: the kernel will allocate the mem-

[3] For instance, when address space is scarce or when fixed addresses are needed.

```
int sio_send(int sock, const void *buf, size_t size, int flags);
int sio_sendto(int sock, const void *buf, size_t size, int flags,
  const struct sockaddr *to, socklen_t tolen);
int sio_sendmsg(int sock, const void *buf, size_t size, int flags);
int sio_recv(int sock, void **buf, size_t size, int flags);
int sio_recvfrom(int sock, const void **buf, size_t size, int flags,
  struct sockaddr *to, socklen_t tolen);
int sio_recvmsg(int sock, const void **buf, size_t size, int flags);
```

Figure 5: Advanced Synchronous Network Interfaces

ory region into which it stores the incoming data. This happens using the kernel-equivalent of the `sio_reserve()` interface. Then the program is notified about the location and size of the incoming data. Until the program is done handling the data the buffer cannot be reused. To signal that the data has been handled, the `sio_release()` interface is used. It is also possible to use the interface to abort the preparation of a write operation by undoing the effects of a previous `sio_reserve()` call.

The `sio_reserve()` and `sio_release()` interfaces basically implement dynamic memory allocation and deallocation. It adds an undue burden on the implementation to require a full-fledged `malloc`-like implementation. It is therefore suggested to require a significant minimum allocation size. If reservations are also rounded according to the minimum size this will in turn limit the number of reservations which can be given out at any given time. It is possible to use a simple bitmap allocator.

What remains to be designed are the actual network interfaces. For the synchronous interfaces we need the equivalent of the `send` and `recv` interfaces. The `send` interfaces can basically work like the existing Unix interfaces with the one exception that the memory block containing the data must be part of a DMA-able memory region. The `recv` interfaces need to have one crucial difference: the implementation must be able to decide the location of the buffer containing the returned data. The resulting interfaces can be seen in Figure 5.

The programmer has to make sure the buffer pointers passed to the `sio_send` functions have been returned by a `sio_reserve()` call or as part of the notification of a previous `sio_recv` call. The implementation can potentially detect invalid pointers.

When the `sio_recv` functions return, the pointer pointed to by the second parameter contains the address of the returned data. This address is in the DMA-able memory area associated with the socket. After the data is handled and the buffer is not used anymore the application has to mark the region as unused by calling `sio_release()`. Otherwise the kernel would run out of memory to store the incoming data in.

For the asynchronous interfaces one could imagine simply adding a `sigevent` structure parameter to the `sio_recv` and `sio_send` interfaces. This is unfortunately not sufficient. The program must be able to retrieve the error status and the actual number of bytes which have been received or sent. There is no way to transmit this information in the `sigevent` structure. We could extend it but would duplicate functionality which is already available. The asynchronous file I/O interfaces have the

same problem and the solution is the AIO control block structure `aiocb`. It only makes sense to extend the POSIX AIO interfaces. We already defined the additional interfaces needed in Figure 2. What is missing is the tie-in with the DMA handling.

For this the most simplistic approach is to extend `aiocb` structure by adding an element `aio_dma_buf` of type `dma_mem_t` replacing the `aio_buf` pointer for DMA-ready operations. To use `aio_dma_buf` instead of `aio_buf` the caller passes the new `AIO_DMA_BUF` flag to the `aio_recv` and `aio_send` interfaces. For the `lio_listio()` interface it is possible to define new operations `LIO_DMA_READ` and `LIO_DMA_WRITE`. This leaves the existing `aio_read()` and `aio_write()` interfaces. It would be possible to define alternative interfaces which take a flag parameter or one could simply ignore the problem and tell people to use `lio_listio()` instead.

The implementation of the AIO functions to receive data when operating on DMA-able buffers could do more than just pass the request to the kernel. The implementation can keep track of the buffers involved and check for available data in them before calling the kernel. If data is available the call can be avoided and the appropriate buffer can be made known through an appropriate event. When writing the data could be written into the DMA-able buffer (if necessary). Depending on the implementation of the user-level/kernel interaction of the DMA-able buffers it might or might not be necessary to make a system call to notify the kernel about the new pending data.

7 Related Interfaces

The event channel mechanism is general enough to be used in other situations than just I/O. They can help solving a long-standing problem of the interfaces Unix systems provide. Programs, be it server or interactive programs, are often designed with a central loop from which the various activities requested are initiated. There can be one thread working the inner loop or many. The requested actions can be performed by the thread which received the request or a new thread can be created which performs the action. The threads in the program are then either waiting in the main loop or busy working on an action. If the action could potentially be delayed significantly the thread would add the wait event to the list the main loop handles and then enters the main loop again. This achieves maximum resource usage.

In reality this is not so easy. Not all events can be waited on with the same mechanism. POSIX does not provide mechanisms to use `poll()` to wait for messages to arrive in message queues, for mutexes to be unlocked, etc. This is where the event channels can help. If we can associate an event channel with these objects the kernel could generate events whenever the state changes.

For POSIX message queues there is fortunately not much which needs to be done. The `mq_notify()` interface takes a `sigevent` structure parameter. Once the implementation is extended to handle `SIGEV_EC` for I/O it should work here, too. One question to be answered is what to pass as the data parameter which can be used to identify the request.

For POSIX semaphore we need a new interface to initiate asynchronous waiting. Figure 6 shows the prototype for `sem_await()`. The first two parameters are the same as for `sem_wait()`. The latter two parameters specify the event channel and the parameter to pass back. When the event reports a successful operation the semaphore has been posted. It is not necessary to call `sem_wait()` again.

```
int sem_await(sem_t semdes, const struct timespec *abstime,
              ec_t ec, void *data);
int pthread_mutex_alock(pthread_mutex_t *mutex, ec_t ec, void *data);
```

Figure 6: Additional Event Channel Users

The actual implementation of this interface will be more interesting. Semaphores and also mutexes are implemented using futexes. Only part of the actual implementation is in the kernel. The kernel does not know the actual protocol used for the synchronization primitive, this is left to the implementation. In case the event channel notification is requested the kernel will have to learn about the protocol.

Once the POSIX semaphore problem is solved it is easy enough to add support for the POSIX mutexes, read-write mutexes, barriers, etc. The `pthread_mutex_alock()` interface is Figure 6 is a possible solution. The other synchronization primitives can be similarly handled. This extends also to the System V message queues and semaphores. The difference for the latter two is that the implementation is already completely in the kernel and therefore the implementation should be significantly simpler.

Along the way we sketched out a event handling implementation which is not only efficient enough to keep up with the demands of the network interfaces. It is also versatile enough to finally allow implementing a unified inner loop of all event driven programs. With `poll` or `select` interfaces being able to receive event notifications for currently unobservable objects like POSIX/SysV message queues and futexes many programs have the opportunity to become much easier because special handling for these cases can be removed.

8 Summary

The proposed interfaces for network I/O have the potential of great performance improvements. They avoid using the most limiting resources in a modern computer: memory-to-CPU cache and CPU cache-to-memory bandwidth. By minimizing the number of copies which have to be performed the CPUs have the chance of keeping up with the faster increasing network speeds.