

# The Case for Reconfigurable I/O Channels

Steven Smith<sup>1</sup>, Anil Madhavapeddy<sup>1</sup>, Christopher Smowton<sup>1</sup>, Malte Schwarzkopf<sup>1</sup>,  
Richard Mortier<sup>2</sup>, Robert M. Watson<sup>1</sup>, Steven Hand<sup>1</sup>

University of Cambridge<sup>1</sup>, University of Nottingham<sup>2</sup>

## Abstract

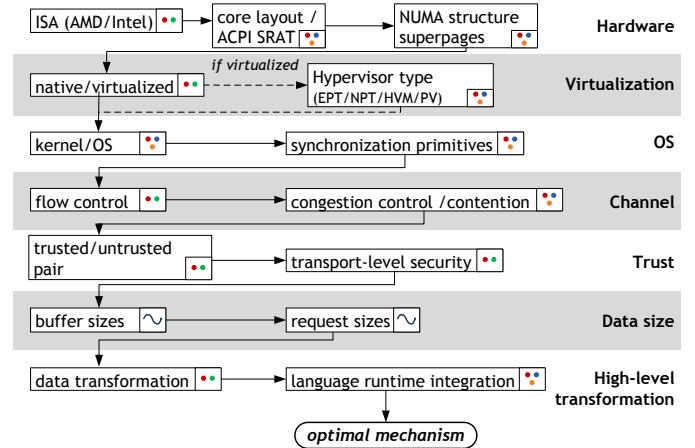
Datacenter environments are increasingly layered, with multicore parallelism, OS virtualisation and NUMA memory all introducing variable latency and throughput for data transmission. For a programmer deploying applications in such a shifting environment, it is unclear how best to use venerable interfaces such as the sockets layer. Kernel hackers realise there is some performance hit to all the software layering, but quantitative figures are hard to find.




This is a position paper of two rather different halves. We first seek to understand just how big the impact of NUMA layouts and OS virtualisation have been on I/O performance. To do this, we implemented a variety of IPC mechanisms (from TCP sockets to shared memory) and benchmarked them under modern multicore hardware and Xen. We discovered a large variance in throughput and latency under different scheduling conditions (over an order of magnitude in some cases), and also some rather inexplicable results which point to the extreme difficulty of predicting cross-layer performance.

In the second half, we describe the early design of a system which aims to overcome these multiplexed I/O scheduling issues. It provides an efficient, zero-copy data transmission interface that automates the selection of the underlying transport, and the facility to dynamically reconfigure transports as system conditions change. Finally, we discuss the implications of extending the OS with explicit I/O flow tracking—eliminating contention, transparent transport-level security and an easy upgrade path to multi-path TCP.

## 1. Introduction

Inter-process communication (IPC) and remote procedure call (RPC) facilities have existed in operating systems for many decades. Ever since the first parallel applications ran on time-sharing machines, programmers have sought ways to communicate between processes running on a single machine, and, equally, the first networked applications introduced the concept of sending a message to trigger a remote action. Today, these primitives are more relevant than ever before: parallel programming within a machine and across machines—an increasingly blurry distinction—relies heavily on facilities to pass data between processes. On a higher level, data-flow frameworks for simple parallel processing of large data



**Figure 1.** Variables that may influence the choice of IPC mechanism. Binary variables are indicated as , N-ary variables as , and continuous variables as .

sets depend on passing data between different tasks, which may run anywhere, including local to a machine, on a networked cluster, or far away in a wide-area “cloud”.

And yet, we are stuck with communication APIs closely coupled to the underlying mechanisms used to implement them: the programmer choice of a communication API constitutes an implicit “buy-in” to a whole set of assumptions about the relative locations of the communicating parties, as well as how the message is to be delivered. Even worse, the implicit trade-offs may not be the same in a different environment, and thus the programmer’s choice of API depends on assumptions about the runtime environment (hardware, software and setup) in addition to the characteristics inherent to an implied mechanism. Finally, the environment may not be stable over time: processes or VMs may migrate during their lifetime, and communication channels may begin or cease to be available.

In this position paper, we present evidence to the fact that this issue is both real and pressing (§2). We show dramatic differences in performance between communication mechanisms depending on locality and machine architecture, and observe that the interactions of communication primitives are often complex and sometimes counter-intuitive (§2.2). Furthermore, we show that virtualisation can cause unexpected effects due to OS ignorance of the underlying, hypervisor-level hardware setup.

Faced with this evidence, we switch to designing a solution, and present an early design of a new API that aims to improve the state of scheduling I/O in a now-typical multi-core, multi-VM, multi-host machine cluster (§3). FABLE is a system for the establishment and management of IPC channels using an system-wide naming service (§3.1.1), with the facility to reconfigure channels

dynamically as system conditions change (§3.1.4). Finally, we discuss some of the architectural implications that result from the addition of a system-wide I/O flow service (§4), and conclude with a call to the research community to pay more attention to low-level communication efficiency and performance when designing large, distributed systems, and to develop better and more user-friendly APIs to support this goal (§5).

## 2. Benchmarking existing APIs

UNIX provides a number of mechanisms for inter-process communications, each with their own benefits and tradeoffs. To give us an idea of their relative performance, we conducted a series of micro-benchmarks.<sup>1</sup> We now show results from these benchmarks—some of which are easily explained, while others are very surprising. We conclude that choosing the optimal transport is a hard problem that warrants an automated solution. In making this argument, we will frequently refer to Figure 1, which gives an overview of some of the variables we discovered to have a significant influence on the optimality of a particular communication mechanism.

We first describe the transports we benchmarked (§2.1), discuss the overall findings—a  $4\text{--}15\times$  difference in throughput between the best and worst performance mechanism on a fairly standard multi-core machine (§2.2) as well as unexpected performance artefacts. We conclude that the performance of systems with so much multiplexing is very unpredictable, and that a more dynamic approach to achieving maximum throughput is required (§2.3).

### 2.1 Mechanisms benchmarked

There is a wide design space of communication mechanisms, and Table 1 summarises those we tried. We began with the conventional socket interface used by most applications (§2.1.1), and then moved onto custom shared memory variants (§2.1.2, §2.1.3 and §2.1.4), and the Linux `vmsplice` mechanism (§2.1.5). Finally, to establish a lower-bound on NUMA performance, we measured the interprocessor-interrupt latencies (§2.1.6).

#### 2.1.1 POSIX sockets

The POSIX sockets API is widely implemented in most multi-process operating systems, and specifies TCP connections, domain sockets and pipes.

TCP sockets are used by applications that potentially communicate outside the chassis; they can of course also be used to communicate with processes running on the local host as this makes it transparent to the application whether it is communicating with a local or a remote process. TCP offers a simple model to application programmers but is, however, somewhat inefficient: the stack continues to run its packetisation, congestion control, and retransmission algorithms, even though these are completely redundant on a localhost loopback connection. We believe that this is responsible for the relatively poor performance which we observed for within-chassis TCP in our benchmarking (§2.2).

Domain sockets avoid many of these problems with the absence of concepts of packetisation, retransmission, or congestion control, while still preserving the usual socket interface. The socket interface is, however, itself a weakness: it fundamentally cannot take full advantage of the potential for sharing memory between processes running on the same physical host, as the interface mandates that data be copied at least once.

Pipes present a broadly similar interface to domain sockets, but are internally implemented in a completely different manner and have surprisingly different performance characteristics (§2.2).

<sup>1</sup> All of these tests are parameterised by many of the decisions in Figure 1 and are available online at: <http://github.com/avsm/ipc-bench>

#### 2.1.2 mempipe

Sockets require at least a single data copy and system call to move data, whereas shared memory offers a high-bandwidth, low-latency alternative when both end-points are on the same OS. Unlike socket communication, there is no standard for shared memory communications, and so we implemented the `mempipe` transport for the purposes of these benchmarks.

The `mempipe` data producer and unconsumed pointers are in memory that is private to the transmitter, while the consumer pointer is in memory that is private to the receiver. Each message has a header flag indicating its freshness and payload length. To receive a message, the receiver waits for the freshness flag to be set in the next header. The message is then consumed, the freshness flag is cleared and the consumer pointer advances.

Transmission requires that sufficient space is available in the ring. The free space is precisely that between the producer and next acknowledgement pointers, and so the transmitter might need to advance the next acknowledgement pointer to ensure that enough space is available. The transmitter waits for the unconsumed-message flag to be cleared in the relevant message header, and then advances the pointer by the size of the acknowledged message. Once the pointer is advanced far enough, the transmitter can send its message. Once the payload has been deposited, the transmitter finds the header immediately *after* the payload area and clears the unconsumed-message flag in that header, before moving back to the header *before* the payload and setting the size and unconsumed-message flag in that header. This two-step process ensures that the receiver does not advance ahead of the transmitter, as if the receiver consumes the message before the transmitter is ready to produce the next one, it will get block waiting for the next message's unconsumed-message flag to be set<sup>2</sup>.

Constructing a good shared memory transport requires making a number of decisions about the end-points. For instance, while there is an transfer of ownership from the transmitter to the receiver in `mempipe`, there is no attempt to enforce it. The transmitter can continue to modify the message payload buffer whilst the receiver is reading it, and so assumes that both sides trust each other to implement zero-copy transmission correctly. We can thus measure the cost of enforcement separately, unlike with the socket API which always enforces that both sides are untrusted. In our benchmarks, the “safe” version of the transport copies the received message into a private buffer before offering it to the application.

#### 2.1.3 Spin-wait vs. futex use

The `mempipe` protocol involves waiting for flags to be set or cleared, for which we consider two schemes. In `mempipe-spin`, the process spins and tests the flag repeatedly. This requires no system calls, locked operations, or memory barriers beyond those implicit in x86 memory accesses, but introduces massive contention for other processes scheduled on the same core. We therefore also consider a version using Linux's `futex` system call. This provides two core operations: block while a given memory location has a given value, and a kernel notification that a memory location has potentially changed. We use these facilities to eliminate all of the spin-wait loops and replace them with blocking operations, at a cost of code portability.

We optimize performance by including an extra flag indicating whether anyone is potentially waiting on a message; this avoids a system call in the wake operation when on the fast path where there are no waiters. Nevertheless, maintaining the flag is itself moderately expensive, and requires atomic operations for every message

<sup>2</sup> Messages are variable size, so is possible that what is now a header was previously message payload, so the header is effectively uninitialised memory until the flag is cleared.

Benchmark	§	Description	Copies
tcp	2.1.1	Standard TCP/IP connection to local loopback.	2
tcp-nodelay	2.1.1	TCP/IP connection with TCP_NODELAY option set, disabling data buffering before sending a packet.	2
unix	2.1.1	Unix domain socket (in streaming mode).	2
pipe	2.1.1	Standard pipe connecting two processes.	2
vmsplice-coop	2.1.5	Sender supplies local page descriptors to a pipe, receiver copies from the sender-allocated memory and notifies.	1
shmem-pipe	2.1.4	-unsafe: meta-data defining extents referring to allocations in a shared-memory region are sent through a pipe. -safe: same as above, except copying the data into a private buffer before inspection.	0 1
mempipe-futex	2.1.3	-unsafe: shared-memory ringbuffer with receiver blocking on a futex awaiting meta-data arrival. -safe: as above, except copying the data into a private buffer before inspection.	0 1
mempipe-spin	2.1.3	-unsafe: shared-memory ring buffer with receiver spinning on meta-data arrival and inspecting data directly. -safe: same as above, except copying the data into a private buffer before inspection.	0 1
ipi	2.1.6	Low-level hardware inter-processor interrupts ( <i>notification-only scheme, data transfer handled seperately</i> ).	N/A

**Table 1.** Different IPC/RPC mechanisms benchmarked.

sent. Atomic operations are themselves quite expensive (although much less so than a system call), and this reduces performance relative to the spinning version of the transport.

The decision of spinning versus futex is further complicated by virtualisation, where the process may have the *illusion* of having a CPU to itself, but is in fact co-scheduled with another VM and would have much lower throughput if spins. The number of system calls also becomes a more important consideration when virtualised, due to extra privilege checks making them more expensive than under native operation. The programmer must thus use different transports depending on their deployment environment.

### 2.1.4 shmem pipe

The `mempipe` futex mechanism has two important weaknesses: it cannot be integrated with existing `poll`- or `select`-based event loops, and messages must be processed entirely in order. These motivated the development of a third shared-memory transport mechanism, `shmem-pipe`. Like `mempipe`, this scheme communicates payload data via a shared memory area, but message metadata is communicated out-of-band using a pair of ordinary POSIX pipes. Rather than managing the shared memory area as a contiguous ring, it is managed using a heap allocator operating in the transmitter. When the transmitter sends a message, it allocates shared space, populates the shared memory using the contents of the message, and sends a small 8-byte descriptor through a metadata pipe. The receiver collects the descriptor and processes the payload in shard memory. Once it has finished with the message, it releases the shared space back to the transmitter by writing another descriptor into the other metadata pipe.

This scheme is zero-copy (like `mempipe`) and can efficiently transfer large amounts of data between processes, and is also easier to integrate with existing `select` loops (as one can `select` on the metadata pipes). Its main disadvantage is that it requires more system calls, even in its fastest case, in order to move the descriptors back and forth through the pipes. Fortunately, these operations can often be batched and sometimes exceed the performance of `mempipe-futex`. This scheme is a userspace equivalent to the Xen virtual device model, where guest kernels transfer shared memory pages using a coordination page for the metadata [11].

### 2.1.5 Cooperative vmsplice

Linux offers a `vmsplice` system call for single-copy IPC, which moves pages into a remote pipe queue (as opposed to `write`, which copies data from the sender into freshly allocated pages in the pipe queue). This data is then copied into the receiver when it calls `read`. Hence, a two-copy pipe is transformed into a single copy one, and makes for an interesting compromise between sockets and our shared memory transports. Unfortunately, `vmsplice` makes it impossible to determine when the page has been read, and the sender can safely reuse it. Thus, the only way of using the interface

with an arbitrary reader is for the writer to unmap the pages after sending, and to then re-allocate additional buffers.

Our `vmsplice-coop` transport fixes this issue by augmenting the main data pipe with an additional metadata pipe that is used by the receiver to tell the sender when it is finished with a given page. Data can thus be copied from the transmitter’s buffers directly into the receiver’s buffers, without a need to modify the receiver and without need to establish a shared-memory region. As with `mempipe`, this relies on the receiver behaving correctly, and therefore cannot be used if the receiver is untrusted.

We hope this brief adventure into shared memory IPC design makes it apparent just how large the design space is! There are many APIs (such as `vmsplice`) that have emerged after Berkeley sockets that encode some, but not all, of these variants.

### 2.1.6 IPIs

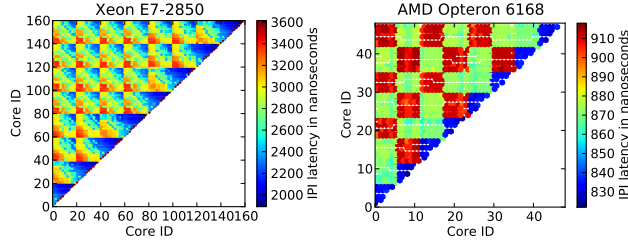
All the schemes described above still involve a considerable amount of software (e.g. in the kernel). In order to examine the hardware layouts much more closely and establish a lower-bound to how efficient inter-process communication can be, we dropped down to the lowest software notification layer—an *inter-processor interrupt*, or IPI.

This is a hardware mechanism which allows a processor to trigger an interrupt in a remote processor. Apart from shared memory, this kind of interrupt-based interface is the fastest mechanism offered by the hardware for inter-processor communication, and so provides an effective lower-bound on the latency of blocking communication primitives.

We test it with a simple “ping-pong” test, where the CPUs involved in the test handle incoming IPIs by generating an outgoing IPI targeted at wherever the incoming interrupt came from. This test is run in a modified Xen without any guest kernels or dom0 running, and so represents the minimum achievable latency with the given hardware.

## 2.2 Benchmark Results

We evaluated our transport mechanisms on an AMD Opteron 6168 (“Magny-Cours”) with a total of 48 cores, split into six cores per die, two dies per socket and a total of four sockets [4]. The machine has 64 GB of DDR3 RAM, and represents a reasonably standard high-end server. We also ran selected benchmarks on other machines in order to get a feel for the effect of different architectures on the results. These machines include an Intel Q6600 (“Kentsfield”) 4-core machine with 8 GB of DDR2 RAM, and an experimental machine with 80 Intel Xeon E7-2850 processors (160 threads). All machines run 64-bit Linux 3.1.0, and Xen 4.1 with the same dom0 kernel in the virtualized tests. The tests were compiled using `gcc 4.6.1` with optimisations enabled (`-O3`). All benchmarks were repeated at least 10,000 times, and both participating cores were pinned to a particular set of CPUs.



**Figure 2.** Comparison of IPI latency between cores on an 80-core Xeon E7-2850 machine and the 48-core Opteron 6168.

### 2.2.1 IPI Latency

The IPI latencies should reveal the NUMA layout of a host, and so we ran them on the 48-core AMD (Figure 2 (*right*)) and the 80-core, hyper threaded Intel (Figure 2 (*left*)). For the AMD machine, each cell in the grid represents a core, with sockets represented by 3x2 blocks. The Intel machine is more complex, with each cell representing a pair of hardware threads, cores represented by 2x2 blocks of threads, and sockets represented by 10x10 blocks of cores. The sockets are themselves divided into 2 trays, each of 4 sockets.

As expected, the fastest inter-core latencies on both machines are observed when communicating with another core in the same socket (the blocks along the diagonal of the diagram), and cross-socket latencies are somewhat higher.

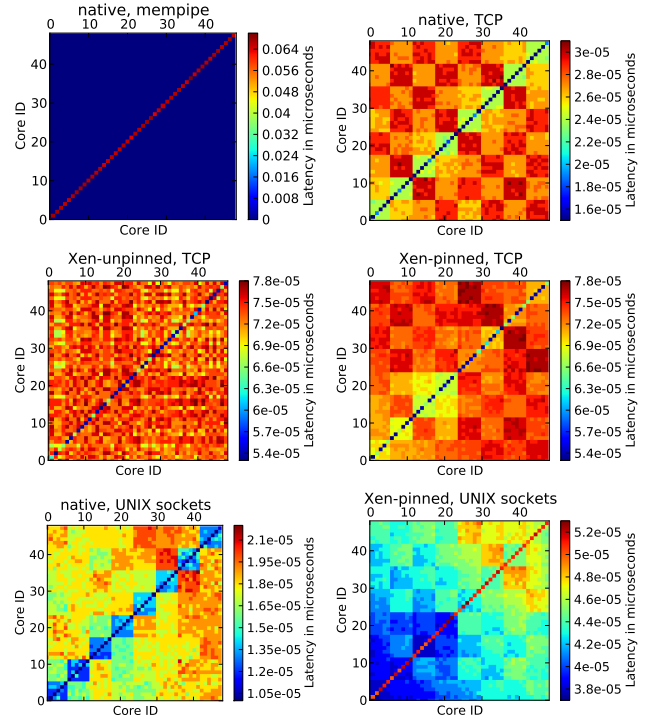
On the Intel machine however, the latencies vary dramatically when talking to a different core on a different socket, showing that there is another factor (or hardware layout) affecting off-socket communications. This complexity is at a layer far below that available to most application developers, and suggests that the choice of transport strategy should be delegated to a system service. We are continuing to investigate the source of this variance, and have mostly ruled out issues such as incorrect CPU enumeration.

### 2.2.2 Latency

Our next benchmarks focus on two key metrics: the inter-core communication *latency* (a “ping-pong” test), and the *throughput* for different message sizes, with one core set up as a producer and one as a consumer. We decided to focus on these aspects and limit the investigation to within-chassis communication both because this case exhibits the most interesting variety of mechanisms, and since within-chassis communication represents a pathological, but increasingly frequent case for today’s data-flow engines.

Figure 3 summarises results of latency measurements on the 48-core AMD machine. Each grid point represents a single test independently run pinned across those cores. The top result shows the performance difference between the shared memory *mempipe* (*left*) and a TCP socket (*right*). As expected, the spinning shared memory is generally far lower latency than TCP, except when communication is scheduled on the same core, at which point it is far worse. The TCP grid clearly reveals the underlying NUMA memory layout of the host, with the on-socket communication being the best latency.

We then ran the same TCP tests under Xen with a single 48-vCPU dom0 (Figure 3 (*mid-left*)). The results are generally more latent than the native case due to the extra system call cost, but also completely obscure the underlying NUMA layout. We traced this to the Xen scheduler continuously rebalancing the vCPU mappings, and tested out the theory by pinning every vCPU to its associated physical CPU (Figure 3 (*mid-right*)). This restored a different NUMA latency graph, since Xen enumerates physical CPU numbers differently from native Linux.



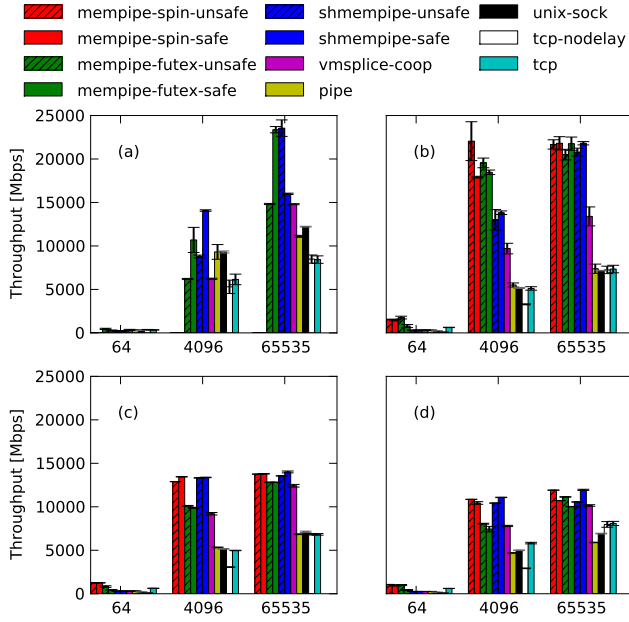
**Figure 3.** Ping-pong latency in different configurations, using the 48 core system described earlier. Xen tests are run in multi-virtual CPU Linux domains running on Xen, while native tests are run in Linux running natively without Xen. Xen-unpinned shows the effect of allowing the Xen scheduler to assign virtual CPUs to physical CPUs; in all other tests, we manually specified this assignment.

We also observed very counter-intuitive results when measuring domain socket latency (Figure 3 (*bottom*)). The native results (*left*) show that the latencies for domain sockets are higher than shared memory, and less than TCP. When run under Xen however, the interacting schedulers have a completely different performance profile, even with the virtual CPUs pinned (*right*). We ran additional tests on Xen to determine where such dramatically different behaviour is coming from, by running the latency tests in a guest microkernel based on the Xen MiniOS. Early results suggest a bug (or at least, odd interactions) between the vCPU numbers and the latency of event channel (i.e. virtual IRQ) latencies.

### 2.2.3 Throughput

We also measured the data throughput for all the transport mechanisms described earlier in Table 1, with individual data request sizes of 64 bytes, 4KB and 64KB. All the tests were configured so that the buffers written were all verified by the reader to contain an incrementing sequence number.

The results are summarised in Figure 4, and show a large variance in performance across transport mechanisms. TCP (with or without Nagle’s algorithm disabled) is slower than all of the other transports, except at 64KB request sizes when it is slightly faster than pipes and domain sockets (due to the large MTU set by Linux when it detects that TCP is going over localhost). The shared memory transports are all clearly better, but there is no clear “winner” between the variants. For example, the spinning version is pathologically bad when both end points are scheduled on the same core, whereas it performs 4-15x faster than TCP across cores. Although not shown here, the throughput results on Xen exhibit significantly



**Figure 4.** Throughput comparison for different mechanisms on a non-virtualized 48-core Opteron 6168. (a) same core, (b) same socket, same die/MCM, (c) same socket, other MCM, (d) different socket, other MCM

more variance. For comparison, we also tested a shared-memory inter domain interconnect (“libvchan”) under Xen, which performed around 30-40% faster than TCP for 4KB request sizes.

### 2.3 Benchmark Summary

We make two key observations after running micro-benchmarks in a variety of different environments: (i) there is no silver bullet for communication between arbitrary processes, only domain-specific solutions. Picking one of these is hard, and the environment in which the decision is made is neither always known in advance, nor is it necessarily stable; (ii) the plethora of variables influencing the performance of a particular mechanism makes the decision unintuitive and lead to unexpected performance effects.

Application developers do not have access to the innards of kernel and hypervisor scheduling, and so cannot make informed decisions about which transport to best use. Most cloud computing providers do not provide the privileged access required to pin vCPUs in guests, and indeed their business models depend on the efficiency gains from statistical multiplexing.

## 3. A Possible Solution: The FABLE Library

We have so far shown that existing APIs are rather unpredictable for high-performance communications due to multiplexing introduced by hardware, hypervisor, kernels and other software layers. Ideally, the programmer should not have to worry about the specific environment they are deploying in, but focus on specifying their data processing needs and let the operating system schedule resources.

We now change tacks from benchmarking and move to the early design of the FABLE library that aims to improve I/O scheduling in the face of all these software layers. It has the following design goals beyond the standard socket API: (i) automatic selection of the best transport protocol between end-points, regardless of if they are on the same kernel, VM or cluster; (ii) support dynamic reconfiguration of the connection when system conditions change;

and (iii) efficient, zero-copy data transmission API that minimises privilege transitions.

FABLE consists of a user-space application library, an extra system call to register with a new name service daemon, and some extensions to existing polling system calls to support the new I/O descriptors. Figure 5 illustrates the following layers:

- **Naming:** all end-points are explicitly named, and a system service (opaque to the library user) tracks the location of processes and virtual machines and notifies them of reconfiguration events. If virtualised or running in a cluster, this name service can register with a higher-level one that has more accurate system-wide knowledge.
- **Connection:** connection setup is similar to POSIX sockets, except that the end-points are named services. Every connection has a single transport mechanism, ranging from tightly coupled shared memory, to a TCP connection, to a page-flipping memory pipe (§2.1). The client specifies if the remote end-point is trusted to cooperate, or if private data copies are required.
- **Flow:** buffers structures for reading and writing are always allocated by the FABLE library, and are tailored for the connection they are associated with (e.g. an entry in a shared memory ring). Buffers are *single-use only*, and buffer creation calls are where back-pressure is applied, rather than at the point of reading or writing. If buffers are unavailable, the application polls to be notified when more are available.
- **Data:** every buffer is owned by exactly one FABLE connection, and ownership is transferred either via a `release` back to the system (e.g. after a read), or via a `commit` to write it onward to the next end-point. Once ownership has been transferred, it can never be regained and new buffers must be requested.

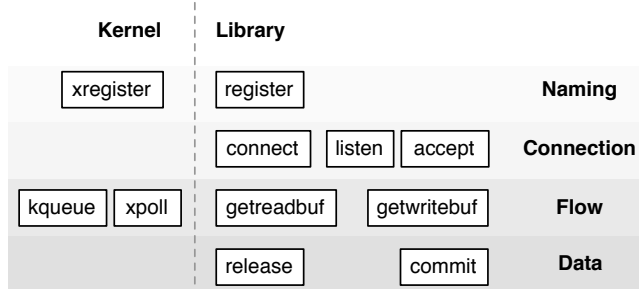
When designing FABLE, we assumed that nested scheduling layers will dominate architectures for some years, due to the popularity of virtualisation and multi-core hardware. The addition of a system-wide name service for end-points is key to keeping track of I/O flows in such complex environments. The FABLE name service is hierarchical and can keep higher-level software layers informed about activity within a particular domain, up to and including a distributed cluster of physical hosts. The ultimate goal is to form an accurate view of dataflow requirements for all applications across a cluster, and provide more structured information for schedulers to maximise I/O throughput across a cluster. We discuss the broader implications of this name service more later (§4).

The FABLE name service is only used to coordinate the establishment of data channels and track their lifetime. Once established, the data transfer between two endpoints is designed to be highly efficient and not require a system call for a read/write operation, although some transports may choose to do so (e.g. remote TCP when using kernel sockets). This is particularly important for throughput in virtualised environments, where system calls can be disproportionately expensive due to privilege checks by the hypervisor.

Although FABLE provides a new API, it can also be integrated directly into existing applications via a socket compatibility layer (§3.2). As we noted earlier, the sockets layer forces at least a single data copy and so is often less efficient, but the facility to track all I/O operations across the system remains extremely useful.

### 3.1 API Overview

We will now describe the lifecycle of an FABLE connection in more detail, starting with the name mechanism (§3.1.1), how connections are established (§3.1.2) and data transferred (§3.1.3), and finally how the reconfiguration process works (§3.1.4).



**Figure 5.** Stages of an FABLE session: naming, connection setup, buffer management, and data synchronisation. Note that normal data operations do not require a system call.

### 3.1.1 Naming

Every FABLE connection is associated with two named end-points.<sup>3</sup> The application calls the `xio_register_name` to register a new end-point, and obtains an opaque `xio_context` structure in return. The library does not keep much state—instead, it accesses a system-wide name daemon via a kernel file-descriptor interface and uses this to register with the name service. This descriptor is used by the name service to track the `xio_context` for its lifetime, including the details of where it is scheduled, and the connections emerging from it.

Most of the policy behind connection handling is implemented in a user-space daemon that listens for FABLE registrations and scheduling changes from the kernel. This daemon is responsible for implementing all the policy for connection rendezvous between end-points, and acts as a system-wide database of I/O flows. When running in on a native kernel on a physical host, it is primarily concerned with ensuring that communicating processes are scheduled close to each other (from a NUMA and core layout perspective). Once virtualised however, it registers with the VM management stack and keeps it informed of the event stream. Similarly, if a host joins a cluster of physical machines and wishes to cooperate with them, the name service can integrate with Zookeeper [6] to handle distributing its local metadata to the other hosts.

Once an `xio_context` has been obtained, it can be used to establish multiple connections to other end-points via `xio_connect`, and also to listen for incoming connections via `xio_listen`. FABLE names are URIs and so the connection API converts a name into a concrete connection, with the application unaware of the precise transport unless it has been explicitly specified in the name. The `xio` schema is reserved for FABLE-aware end-points, and some other schema such as `tcp` or `udp` are supported to facilitate external communication via standard protocols (and are needed for the socket emulation library).

### 3.1.2 Connection

Connection establishment requires both end-points to agree that they wish to communicate (i.e. that one is in a `listen` mode and the other is connecting), and the selection of a transport mechanism that is agreeable to both ends. Since the FABLE name service has both of the services registered, it acts as the intermediary and calculates the best transport protocol for the two end-points. A successful `xio_connect` library call will return an opaque `xio_handle` that is used to reference the connection by the application.

The details of transport selection are necessarily quite complex, since they depend on some static factors (hardware memory and

<sup>3</sup> Although we have considered mechanisms for multicast, the first version does not include support for it.

API call	Description
<code>xio_register_name</code>	Register this end-points name
<code>xio_connect</code>	Establish a connection to a remote end-point
<code>xio_getreadbuf</code>	Obtain a buffer with data from the remote
<code>xio_getwritebuf</code>	Allocate a buffer for writing to the remote
<code>xio_release</code>	Release a buffer back to the system
<code>xio_commit</code>	Commit a buffer for writing to the remote
<code>xpoll</code>	Poll a POSIX fdset and a FABLE connection set

**Table 2.** List of main FABLE API calls.

core layout) and dynamic factors (e.g. virtualisation introducing external load). The system name service is thus better-placed to make this decision, instead of the application itself.

### 3.1.3 Data Transmission

Applications never allocate their own I/O buffers, and instead obtain buffers using the `xio_getreadbuf` and `xio_getwritebuf` calls. This allows FABLE to allocate optimal buffers for the transport associated with the connection—e.g. low-memory if the network card requires it for DMA, or from a shared memory segment on the closest NUMA node, or with space reserved for TCP/IP packet headers. These are optimisations reminiscent of exokernels [5] and explicit path selection [9] that have so far not found their way into mainstream UNIX-like systems.

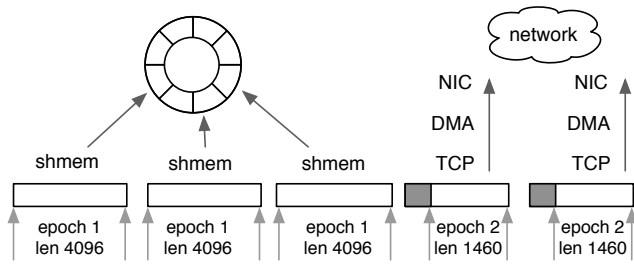
Buffers are very similar to `iovec` structures, and include a pointer to the I/O memory and its size. They also include a reference to the `xio_handle` that created them, and an epoch number to help with reconfiguration (§3.1.4). An important property of buffers is that they are *single use*, and cannot be reused once they are freed or written to another end-point.

An `xio_getreadbuf` call is non-blocking, and returns an array of buffers that are filled with data, or an empty set to indicate that the application should poll for more data. When an application is finished with a buffer, it must call `xio_release` to hand it back to the system. Since there are a limited set of buffers associated with each connection, the `xio_getreadbuf` call can return an `ENOSPACE` to indicate that the application is holding onto too many read buffers and should release some before requesting more. To prevent deadlock, the application may handle this by copying read buffers into private memory and releasing them early.

The write data path calls `xio_getwritebuf` with an optional parameter to specify the maximum size of the available data. This returns an array of buffers which should be filled in any order by the application. The size of each individual buffer is very transport-specific, and ranges from a 4K page size for shared memory, to slightly smaller than an interface’s MTU for TCP (to reserve space for packet headers). When a buffer is filled for writing, the `xio_commit` call will transfer ownership of the buffer back to the library, which queues it for writing. The application may no longer modify or access this data once it has been committed—this is advisory if the connection is trusted, and otherwise enforced via a private copy being taken by the receiver or the page reference being unmapped from the transmitting end.

The notion of single ownership of buffers is key for constructing efficient stream processing engines, where processes perform a combination of data processing and proxying. For example, consider a web server that reads pages from disk via one FABLE channel, transmits the disk pages to a `memcached` process, which then serves it to a network interface. The connection from the disk layer will be a set of page-aligned buffers, whereas the connection to the memory cache is a large shared memory ring. In this situation, the application may commit a read buffer from the disk channel *directly* into the `memcached` channel, despite the disk buffer not being obtained from the memory cache channel. Every buffer tracks its home connection, and so the FABLE library performs the appro-





**Figure 6.** FABLE buffers are variable-length and marked with an epoch to separate reconfiguration events. The above example shows a shared memory connection being switched over to zero-copy TCP, with header-space reserved to facilitate DMA.

priate translation to convert between transport mechanisms (usually via a slow copy). Once the foreign buffer has been committed, the upstream writer is responsible for releasing it.

Astute readers will observe that the end-to-end data path in the previous example is not very optimal, as a page-aligned data source (the disk) and a page-aligned data sink (the network card) are being interrupted by a data copy on a shared memory channel to the intermediate cache process. This may be fine if the data is being changed along the way, but large file transfers would be more efficient if the memory cache switched its transport mechanism to also be page-based. *Channel reconfiguration* lets FABLE perform precisely this optimisation, and is described next (§3.1.4).

### 3.1.4 Reconfiguration

Every `xio_handle` also has a file descriptor that can be obtained to poll for reconfiguration events. A reconfiguration indicates that the underlying transport mechanism is being changed, and that the application should drain any older buffers as quickly as it can (either by releasing them, or committing them for a write). The *epoch number* in each buffer is used to distinguish between the different transport mechanisms (see Figure 6).

While the reconfiguration notice to the application is an synchronous, the actual change is very asynchronous and similar to Xen live migration [3]. The new transport data path is established first, without altering the existing one. A notification is then sent to the FABLE library instance via the event file descriptor associated with the end-point. All new buffers requested by the application are now associated with the new transport, and the application is given a timeout period to use the old buffers. If the application fails to drain them in time, FABLE can slowly proxy the old buffers to the new transport if it can, or simply terminate the connection.

A good example of the need for reconfiguration is when using a virtual machine cluster. When two VMs are on the same physical host, they establish an inter-domain shared memory communication (e.g. via *libvchan* in recent versions of Xen). If one of the VMs then live migrates to a different physical host, this connection would normally be terminated. With FABLE however, the live relocation is observed by the cluster name service and triggers a recalculation of the transport protocol, and configures TCP instead. The example flow of buffers can be seen in Figure 6.

Aside from this, many of the performance anomalies we observed earlier can be adjusted for via a reconfiguration process. For instance, if two cores are idle and not virtualised, then a low-latency spinning transport may be the most efficient. If another end-point is subsequently scheduled onto the same cores, they will begin contending, and the transport should be reconfigured to a futex-based version. Similarly, if a process is rescheduled to a different NUMA

node, this can trigger the reallocation of memory buffers to ones from the new NUMA node.

There is some resource cost associated with reconfiguring a channel, and it is not intended to be done extremely regularly. Instead, the FABLE name service observes all I/O flows on the system and can be configured to either automatically balance them (e.g. using Kalman filters to smooth out changes [7]), or allow a system administrator to optimise it manually if desired. Either policy is easy to implement due to the existence of the system name service to aggregate and coordinate any reconfigurations.

## 3.2 Integration with Applications

The POSIX socket layer decouples name resolution from connection setup, and the FABLE socket layer bridges them together by caching the results of the name lookup and using that information to translate subsequent socket API calls from the application into the FABLE equivalents. It is mostly prototyped as a user-space LD\_PRELOAD library.

The easiest name resolution call to support is `getaddrinfo`, which accepts a destination name and service, and returns an address family (usually `AF_INET` or `AF_INET6` for IPv4 or IPv6 addresses respectively), and a concrete list of `sockaddr` structures. The FABLE wrapper registers the requested name/service pair with the name service as a “partial lookup”, obtains a handle in return, and returns this as an `AF_FABLE` address family.

The application can then use the returned `sockaddr` through to the usual `socket` and `connect` or `listen` calls. The FABLE wrapper functions use the partial handle to reconstruct the name and service at connection time, and perform a FABLE call if it is local or a normal lookup for external requests. Conveniently, the `getaddrinfo` interface also has a declarative set of hints which indicate the application’s willingness to deviate from standard protocols such as TCP or UDP, and so give a clear indication to FABLE about how much freedom it has to specialise the underlying transport stream.

Supporting the older `gethostbyname` resolution is trickier, since it returns a list of names, and the application must construct a suitable `sockaddr` structure with a destination port filled in. This can be still be handled in mostly the same way as `getaddrinfo`, except that instead of returning an `AF_FABLE` address, a localhost “cookie” IPv4 `sockaddr` is generated that encodes the name handle. The decision about whether to use a FABLE connection is then taken at connection time, by using the fake IPv4 address and port to lookup the partial handle from the name service, and determine if the target address is another FABLE end-point or a connection to the outside world.

## 4. Discussion

The FABLE design augments the operating system with a service that tracks I/O flows for their lifetime, and the facility to reconfigure them. We consider some of the implications of this now.

### 4.1 Upgrading Transport Protocols

I/O contention, in both the network and disk interfaces, is a serious problem in many modern environments, especially those making heavy use of virtualization, and significant research effort has gone into handling these issues [1, 10]. The FABLE name service, with its global knowledge of communication patterns, is well-placed to manage these issues. At the most basic level, this could be as simple as selecting a communication channel which is appropriate to the communication environment. This might, for instance, mean enabling multi-path TCP on some channels, and deciding which paths to use. These decisions would be constantly re-evaluated by FABLE as the communication environment changes, and, where appropri-

ate, channels would be reconfigured; the APIs presented here allow this to be performed transparently to the overlying application. More interestingly, FABLE can integrate with computation schedulers, at both host and cluster level, to schedule communicating tasks in a way which minimises contention. Rather than simply reacting to or tolerating contention, FABLE could in many cases prevent it from even occurring. This should allow more efficient use of existing computation resources.

## 4.2 Transparent Security

Live migration [3] allows VMs to be moved from one host to another, potentially over the wide area network. The VM's network connections generally migrate with it, but this can lead to security issues if a connection that was previously local to a trusted network or physical host is redirected over an untrusted network such as the Internet. The usual approach to this problem is to require that any connection carrying sensitive information which might be so migrated be always encrypted. Unfortunately, cryptographic protections are computationally expensive, and this is inefficient when the communicating VMs happen to be connected via a trusted network (or are even located on the same physical host).

The FABLE name service provides a natural solution: reconfigurable channels (§3.1.4) allow cryptographic protections to be negotiated, upgraded, and downgraded where appropriate, completely transparently to the application, and so ensure that efficient unencrypted communication is used precisely where it is safe to do so. The user-space name daemon is also an appropriate place to store public keys for end-points and a policy describing which networks are considered trusted, and upgrade to efficient new secure transports such as TCPcrypt [2]. FABLE could also support more exotic threat models, by, for instance, enabling cryptographic protection for host-internal shared-memory communication when the host chassis is open.

## 4.3 Exokernels for the Virtualised Masses

Exokernels [5] eliminated multiplexing by exposing I/O layers as libraries, and structuring the application to directly link against them. The problem with deploying exokernels more widely has been two-fold: the difficulty of implementing device drivers, and the lack of a coordination layer to manage such vertical application stacks.

Recent exokernels such as Mirage [8] run directly under Xen to solve the device driver problem. However, they are still awkward to configure and use, since running a VM is far more inefficient than simply forking a process. We envision that FABLE will be useful to not only communicate efficiently between processes and kernels, but also to set up device channels to virtual machines. This not only eliminates much redundant code between Linux and Xen, but also provides the opportunity to optimise I/O channels more deeply as nested virtualisation takes off.

## 5. Conclusion

In this paper, we firstly performed some quantitative measurements of the behaviour of various transport mechanisms, when running on virtualised and multi-core hardware. The performance properties of modern interprocess communication facilities are complex and counterintuitive, and can change dynamically while programs are running. This makes it difficult to design systems which combine both high performance and flexibility.

Secondly, we discussed the early design for FABLE, a system service that dramatically simplifies these issues, allowing developers to straightforwardly produce software which adapts to its communication environment, ensuring efficient communication without sacrificing generality. By providing a single point of global

knowledge of I/O flows, FABLE also allows other components of an interconnected system to optimise their own performance, by, for instance, employing a scheduling policy which interacts well with the actual IO activity observed.

We believe that extending operating systems with a first-class I/O flow scheduler may be key to unlocking the potential of exciting new technologies such as multipath communication, ubiquitous transport encryption, and exokernel-like application specialisation.

## Acknowledgments

We would like to thank David Scott from Citrix Systems R&D for useful discussions and hardware, and Balraj Singh, Ben Laurie and Jon Crowcroft for their comments.

## References

- [1] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010), SIGCOMM '10, pp. 63–74.
- [2] BITTAU, A., HAMBURG, M., HANDLEY, M., MAZIÈRES, D., AND BONEH, D. The case for ubiquitous transport-level encryption. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 26–26.
- [3] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd Symposium of Networked Systems Design and Implementation* (May 2005).
- [4] CONWAY, P., KALYANASUNDHARAM, N., DONLEY, G., LEPAK, K., AND HUGHES, B. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro* 30, 2 (Mar. 2010), 16–29.
- [5] GANGER, G. R., ENGLER, D. R., KAASHOEK, M. F., BRICEÑO, H. M., HUNT, R., AND PINCKNEY, T. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.* 20 (February 2002), 49–83.
- [6] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX ATC'10, USENIX Association.
- [7] KALYVIANAKI, E., CHARALAMBOUS, T., AND HAND, S. Resource provisioning for multi-tier virtualized server applications. *Computer Measurement Group Journal (CMG Journal '10)* 126 (2010), 6–17.
- [8] MADHAVAPEDDY, A., MORTIER, R., SOHAN, R., GAZAGNAIRE, T., HAND, S., DEEGAN, T., MCAULEY, D., AND CROWCROFT, J. Turning down the LAMP: Software specialisation for the cloud. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (Boston, MA, USA, June 2010), USENIX.
- [9] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation* (New York, NY, USA, 1996), OSDI '96, ACM, pp. 153–167.
- [10] RAICIU, C., BARRE, S., PLUNTKE, C., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference* (2011), SIGCOMM '11, pp. 266–277.
- [11] WARFIELD, A., FRASER, K., HAND, S., AND DEEGAN, T. Facilitating the development of soft devices. In *Proceedings of the 2005 USENIX Annual Technical Conference (General Track)* (April 2005), USENIX, pp. 379–382.