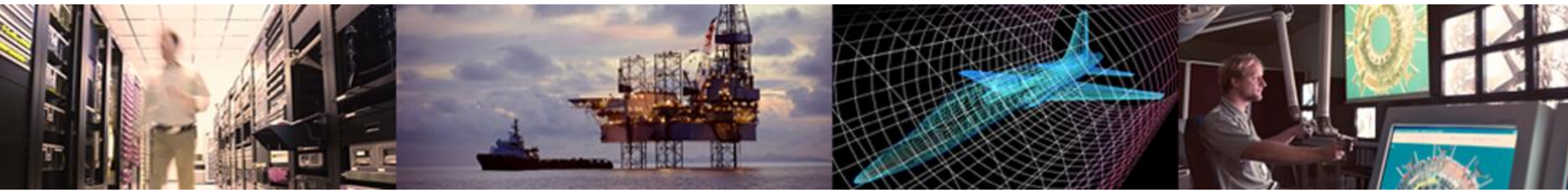
White Paper

Introduction to OpenOnload—Building Application Transparency and Protocol Conformance into Application Acceleration Middleware



Steve Pope, PhD
Chief Technical Officer
Solarflare Communications

David Riddoch, PhD
Chief Software Architect
Solarflare Communications

Solarflare's OpenOnload® application acceleration middleware is an accelerated network stack. It is an implementation of TCP and UDP over IP which is dynamically linked into an application's address space and granted direct access to accelerated network hardware. The network stack interposes network operations from the application and enables them to be handled completely at user-space. In so doing, it bypasses the operating system and significantly improves performance through the removal of disruptive events such as context switches and interrupts which otherwise reduce the efficiency by which a processor can execute application code. This acceleration is most pronounced for applications that are network intensive, such as:

- Market data and high frequency trading applications
- Physical modeling applications such as computational fluid dynamics (CFD)
- Video streaming
- Distributed object caches (or databases) such as Memcached
- Other system hot spots such as lock managers or forced serialization points

OpenOnload dynamically links with an application at run-time and by implementing the standard BSD sockets API, enables an application to be accelerated without modification. Figure 1 illustrates OS bypass in an abstract form, where OpenOnload represents a protocol implementation suitable for direct access to virtualized NIC hardware.

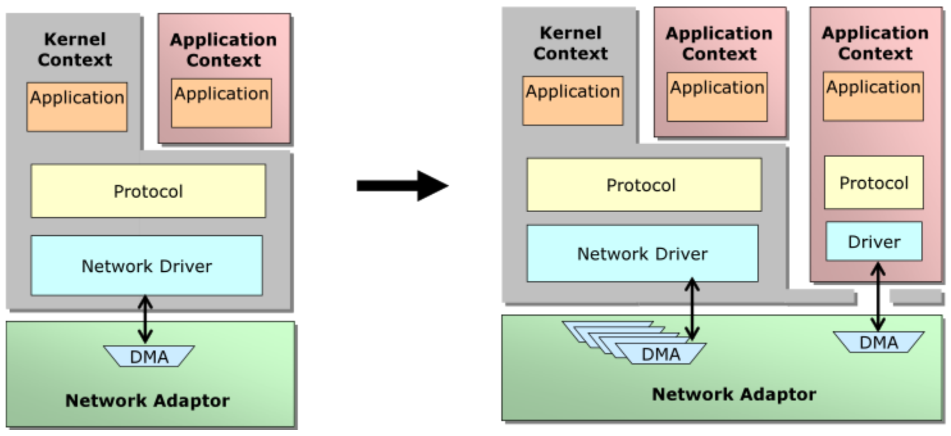


Figure 1

This paper is intended to provide an introduction to the OpenOnload internal architecture and the techniques employed to ensure that **full application transparency** is delivered with high-performance. It should be read in conjunction with the talk, slides and other material on the openonload.org project site.

ARCHITECTURE

OpenOnload is a **passive library,** which means that no threading model is imposed on the application and that the library will work with any language binding. It is common for applications implemented using languages such as C, C++ and Java to be accelerated using OpenOnload. This property also means that the library can operate with the absolute lowest overheads, since protocol processing may take place directly in the context of the thread invoking the networking operation. Especially on receive, the OpenOnload library will generally operate lazily, in that protocol processing does not take place until a calling thread enters the library. This is known as **lazy-receive** processing and has significant benefits to performance, particularly improving processor cache spatial and temporal locality.

There are circumstances when asynchronous protocol processing should take place, for example, when an application thread is not provided for some significant period of time, or when an application exits before all its connections have been closed. For this reason, OpenOnload is a **hybrid** stack, capable of operating at user-space and kernel-mode for any given network flow and able to choose, dynamically, whichever is appropriate. Asynchronous operation is provided by the kernel—typically in response to an interrupt—and provides a robust mechanism to ensure that the OpenOnload network stack responds to protocol events in a timely manner. A pure user-space implementation by contrast would not be able to make such guarantees, since otherwise once an application exits or crashes, all user-space protocol state is destroyed.

Hybrid stack operation is also beneficial for some workloads where there are many more application threads than physical CPU cores. Here the system must necessarily schedule between threads and it is often useful for some degree of background processing to take place in order that timely responses to synchronization operations such as poll(), select() or epoll() may be made. The use of background processing in the kernel context often enables post-protocol processed results to be indicated to the user-space library with lower latency than would otherwise be possible. This feature is important for protocols such as TCP where, for example, the semantics of TCP mean it is not sufficient to simply indicate that a packet has received in order to indicate that a file descriptor has data ready. Hybrid processing also enables significant performance gains to be made for highly-threaded applications, especially if the application is bursty. It is often the case that once a thread is scheduled with a set of active sockets, a number of network operations can be performed in short order. These operations can take place completely in user-space during the time-slice available to the thread. This property remains true even if the stack had been previously operating in kernel mode for some or all of these sockets. The mechanism by which this hybrid operation is enabled is a protected memory mapping from the user-space library onto some of the protocol state associated with each socket. Importantly, this protocol state canonically resides in the kernel and is accessed by the user-mode library component with low overhead via the memory mappings.

The POSIX API performs networking operations via file descriptors. OpenOnload accelerates these operations by **interposing the file-descriptor table, a copy of which is** maintained at user-space. The OpenOnload library is able very cheaply to determine whether or not any given file descriptor should be handled by itself, or else by the regular kernel stack. Operations on sets of pure kernel stack based file descriptors are simply forwarded by the library to the kernel, whereas **operations on mixed sets of descriptors** containing both OpenOnloaded and kernel stack descriptors are handled by the OpenOnload library.

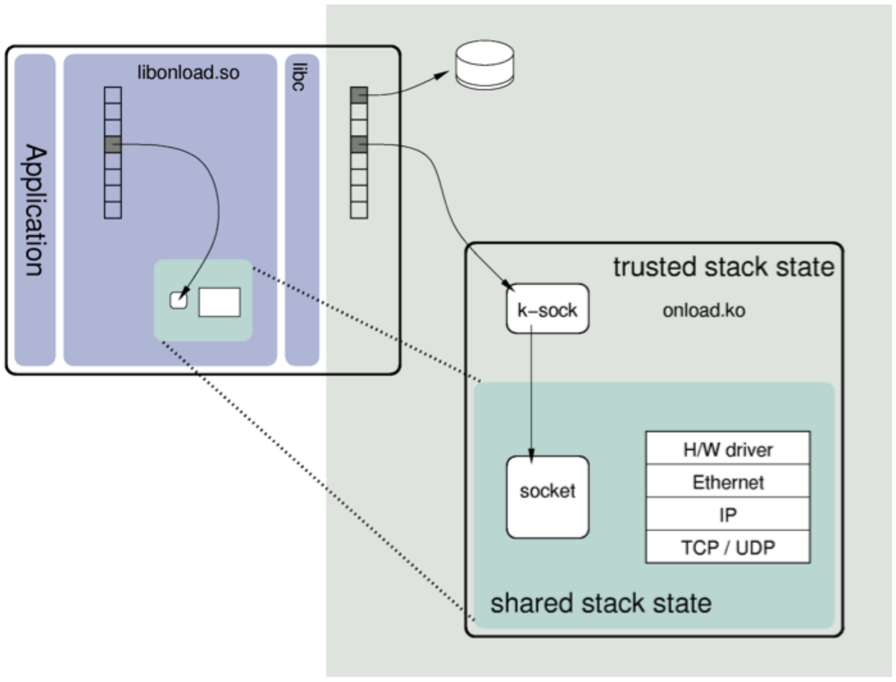


Figure 2

As shown in Figure 2 above, each file descriptor which has been interposed by OpenOnload is backed by kernel state, including a real kernel socket. This kernel socket enables the OpenOnload stack to request resources, such as ports, which are maintained by the native kernel resident networking stack. This feature also enables file descriptor based semantics at the POSIX API to be correctly implemented. For example, over a fork()/exec() operation, the new child process may inherit no state other than open file descriptors from its parent. ie. The child process should have access to the same sockets as the parent, but has lost all of its user-space mappings and state. Correct operation is implemented utilizing a side effect of the property that protocol state is defined to reside in the kernel and therefore, appropriate portions may be mapped into user-space as required.

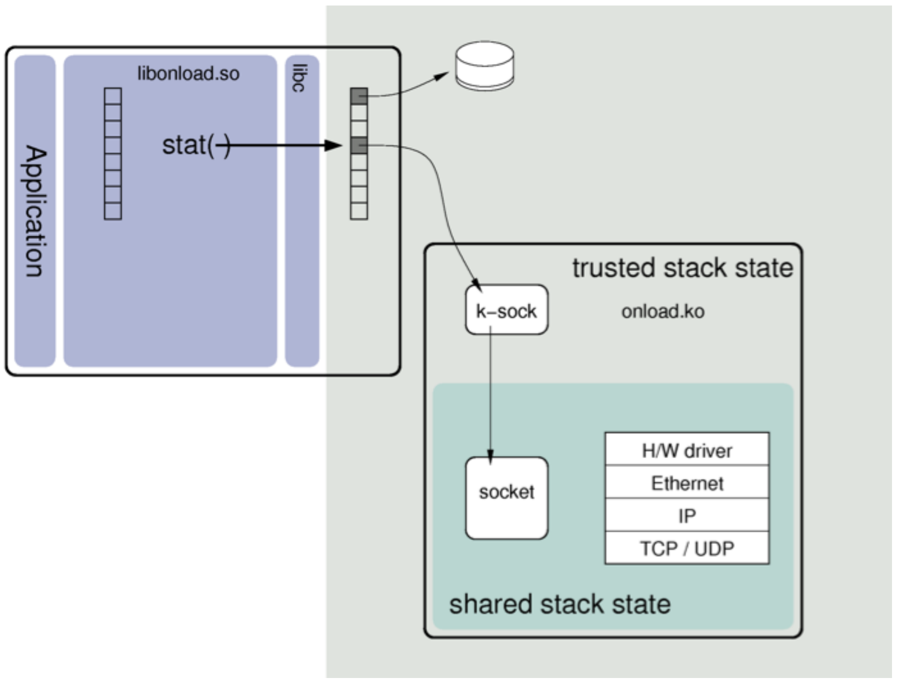


Figure 3

Figure 3 illustrates that following the exec(), the new child process will inherit the environment of the parent and so will dynamically link against and initializes a fresh instance of the OpenOnload library (libonload.so). This new instance does not require priming with any state contained within the parent process' library. Instead OpenOnload will perform a stat() operation on any unknown file-descriptor which it has been passed. The result of the stat() provides a means for the OpenOnload library to discover and recover a memory mapping onto the shared stack protocol state. It is also the means by which other **shared operations on sockets are possible**. Any given socket may be mapped onto a particular OpenOnload network stack instance, allowing for example, **multiple processes to subscribe to and receive data from the same IP multicast groups** whilst accelerated and for maximum performance without requiring multiple packet deliveries from hardware over the PCI bus. Control of this feature is provided both by environment variables at the granularity of the process, or using a programmatic API for socket granularity. This is illustrated by Figure 4, where multiple processes are sharing and able to access the same underlying sockets at user-space.

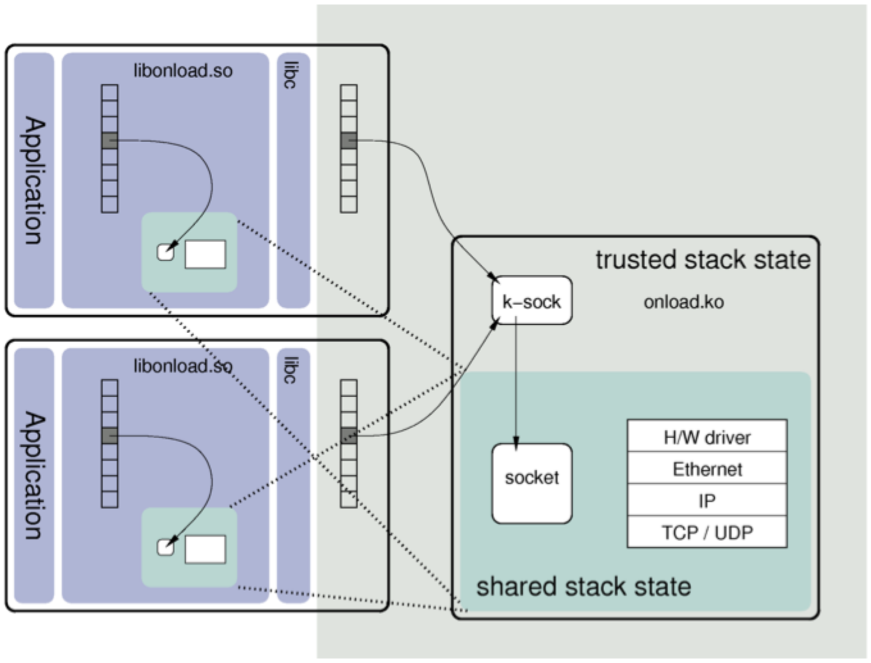


Figure 4

The arrangement shown in Figure 4 should not be applied for applications where an indirect linkage through the shared memory segments is not appropriate. For this reason, the OpenOnload default behavior is not to share a stack following operations such as fork() and exec(), or where multiple processes subscribe to the same multicast groups. Unless specified as indicated above, the default arrangement following fork()+exec() is shown in Figure 5. In this case network operations are not intercepted in user-space, and as a result processing takes place in the context of the kernel for that particular socket.

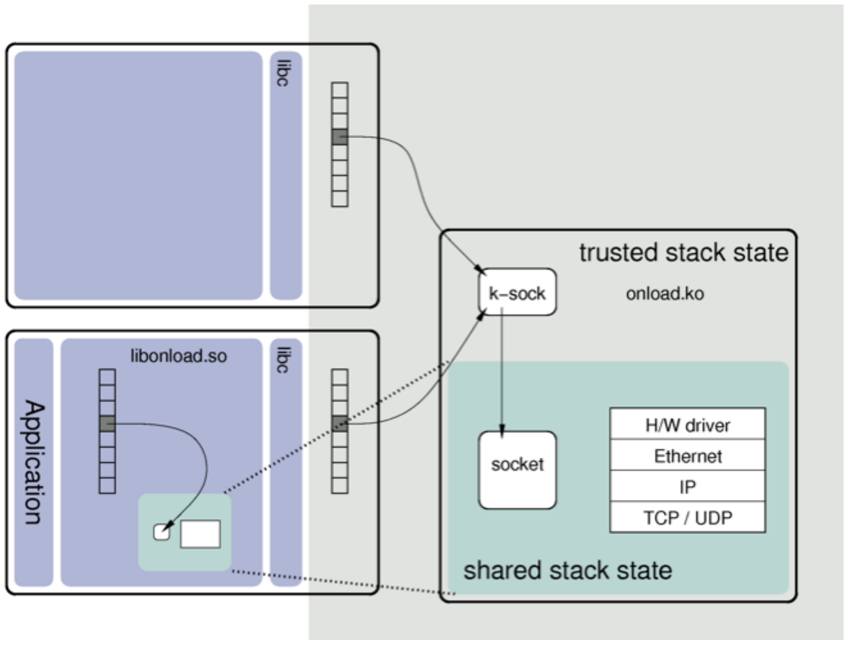


Figure 5

Having a kernel socket backing each OpenOnload enabled socket also allows for mixing OpenOnload-enabled network interfaces and other network interfaces. For example, a UDP socket may receive datagrams from any Ethernet interface. Those which are received from an OpenOnload enabled network interface are delivered directly through the OpenOnload stack, at user-space, whereas those delivered through a non-OpenOnload capable interface are delivered through the regular kernel stack via the kernel socket. The OpenOnload library will cleanly merge such flows, again preserving application transparency.

As well as data operations, significant care is taken to ensure that **multi-tier operations are possible for synchronization** operation such as poll()/select()/epoll(). When a synchronization operation is requested for a file descriptor set containing only OpenOnload enabled file descriptors, the library is able to perform the operation entirely at user-space (and if requested block by busy-waiting for lowest-latency). However, if the file-descriptor set contains mixed file-descriptors, then the OpenOnload library cannot determine the state of the non-OpenOnload file descriptors without checking via the kernel maintained state of the native protocol stack (typically via a system call). In such cases, the library will balance the requirements to deliver correct and timely results from the kernel based file-descriptors with low-latency requirements from the OpenOnload enabled file descriptors by employing heuristics to always check the file-descriptors which can be resolved at user-space, with periodic checking of the descriptors requiring kernel operations. Combined with busy-waiting at user-space, this can be a valuable tool for applications containing mixed file descriptor sets. Hints can be provided to enable the heuristics to be further tuned – for example, an assertion that a particular socket will only receive datagrams from an OpenOnload enabled interface can be very useful to the library. The OpenOnload library has been designed to scale well with very large numbers of sockets and many different thread models.

Other techniques have been employed to ensure the best possible performance with OpenOnload. For example, attention has been paid to reducing lock contention when accessing the library. Many common operations such one thread reading, another writing a socket can take place without any lock contention at all. But where locking is necessary, the developers have spent time to employ techniques such as CASL (or lockfree) data structures in conjunction with advanced deferred work techniques to deliver the absolute best in performance, not only for a simple single-threaded micro benchmarks, but also for the complex interactions of production applications.

To avoid the requirement of managing OpenOnload enabled sockets as a separate network interface, an abstraction is maintained of a single physical network interface for which particular network flows are accelerated onto a virtual interface mapped into a user-address space. Therefore management of the physical network interface is performed using the same tools and operations as are used for regular kernel mode networking. The OpenOnload kernel component registers itself with the Linux kernel control plane and receives full notifications of all network related state (and changes thereof) including properties such as VLAN and teaming status, the ARP cache, ICMP notifications, and the IP routing tables. As a result OpenOnload is able to correctly determine the correct operation for any API call with complete transparency. The control plane information which is received by the OpenOnload kernel component is mapped to the user-space libraries by a read-only memory mapping, enabling user-space access to critical state changes with very low overhead.

The OpenOnload TCP implementation was developed from the ground up, specifically for hybrid user-space/kernel operation. Low-latency for an eager thread is provided by significant layer integration work within the code-base. Lazy-receive techniques are extensively deployed and the stack implements a complete set of RFCs including those implementing advanced congestion avoidance and recovery, such as New Reno with Fast Recovery, SACK and TCP Extensions for High Performance. These features ensure robust and fully compliant performance while maximizing performance. OpenOnload complies with at least the following RFCs: 791, 792, 793, 768, 896, 1122, 1191, 1323, 2001, 2018, 2525, 2581, 2582, 2883, 2988, 3390, 3465, 3708, and is regression tested using industry standard test tools.

OpenOnload represents over 25 man years of development and was architected from the ground up to fully support mixed protocol operation. It is a mature and complete product tested and in full production in many demanding applications, delivering robust performance, ease of deployment with the absolute best in performance.

IN SUMMARY

OpenOnload is implemented using a language agnostic, passive library. This property means that the product may be used in conjunction with any language binding and with other libraries and Middleware.

OpenOnload is a hybrid user-space / kernel stack. This property means that protocol processing can take place at the appropriate place for the application’s needs and enables the stack to support the full POSIX semantics expected by true application level transparency.

OpenOnload operates by interposing the file-descriptor table maintained at user-space. It is able, very cheaply, to determine whether or not any given file descriptor should be handled by the Openonload library, or else by the regular kernel stack.

OpenOnload is able to operate efficiently with mixed sets of file-descriptors including multiple protocols and descriptors from arbitrary network interfaces

OpenOnload enabled sockets are backed by a real kernel socket. This enables the OpenOnload stack to request resources which are maintained by the regular kernel stack, such as ports. It also enables file descriptor based semantics at the POSIX API to be correctly implemented.

OpenOnload supports shared operations on sockets. This enables multiple processes to subscribe to the same multicast groups whilst accelerated. Enables fork()/exec() semantics and debugger applications to attach to protocol state.

OpenOnload supports multi-tier processing of synchronization operations, such as poll()/select()/epoll().
This enables efficient processing of mixed kernel / OpenOnload file descriptor sets.

OpenOnload utilizes lock-free data structures and deferred work techniques to ensure best performance not only for simple single-threaded micro benchmarks, but also for the complex interactions of production applications with multiple threads.

OpenOnload's kernel module registers with the Linux kernel control plane and receives full notifications of all network related state. This enables OpenOnload behave in the same way as the kernel stack, and with complete transparency. Read only memory mappings reflect the control plane to user space, enabling low overhead operation.

OpenOnload TCP was designed specifically for hybrid user-space operation and supports both high-performance, and fully conformant operation in congested environments by implementing a complete set of RFCs including advanced congestion avoidance and recovery.

ABOUT THE AUTHORS

Steve Pope is the CTO of Solarflare. Previously he co-founded Level 5 Networks and prior to that was a post-doctorate researcher in the field of high-speed networks and operating systems at Olivetti Research Labs, which later became AT&T Laboratories Cambridge. He holds a PhD in Computer Science from the University of Cambridge.

David Riddoch is the Chief Software Architect of Solarflare, which he co-founded in July 2002. Previously, David was the architect and lead developer of the software for the CLAN high performance network project at AT&T Laboratories Cambridge. David holds a first class degree in computer science and a Ph.D. in high performance networking from the University of Cambridge.