DLibOS: Performance and Protection with a Network-on-Chip

Stephen Mallon
University of Sydney
Sydney, NSW, Australia
smal8373@uni.sydney.edu.au

Vincent Gramoli
University of Sydney
Sydney, NSW, Australia
vincent.gramoli@sydney.edu.au

Guillaume Jourjon
Data61-CSIRO
Sydney, NSW, Australia
guillaume.jourjon@data61.csiro.au

Abstract

A long body of research work has led to the conjecture that highly efficient IO processing at user-level would necessarily violate protection. In this paper, we debunk this myth by introducing *DLibOS*, a new paradigm that consists of distributing a library OS on specialized cores to achieve performance and protection at the user-level. Its main novelty consists of leveraging network-on-chip to allow hardware message passing, rather than context switches, for communication between different address spaces.

To demonstrate the feasibility of our approach, we implement a driver and a network stack at user-level on a Tilera many-core machine. We define a novel asynchronous socket interface and partition the memory such that the reception, the transmission and the application modify isolated regions. Our high performance results of 4.2 and 3.1 million requests per second obtained on a webserver and the Memcached applications, respectively, confirms the relevance of our design decisions. Finally, we compare DLibOS against a non-protected user-level network stack and show that protection comes at a negligible cost.

CCS Concepts • **Computer systems organization → Multicore architectures**; • **Networks → Network on chip**; • **Security and privacy** → Operating systems security;

Keywords Performance; Latency-critical applications; Network-on-Chip; Microsecond-scale computing; Memory protection

ACM Reference Format:
Stephen Mallon, Vincent Gramoli, and Guillaume Jourjon. 2018. DLibOS: Performance and Protection with a Network-on-Chip. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS '18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3173209

1 Introduction

The microseconds era requires novel architectural approaches throughout the system software stack. NVMe SSDs have read latencies of tens of microseconds whereas network devices within datacenters have reached latencies of the order of microseconds [3]. This is in contrast with current storage and networking stacks that add significant overheads and assume IO (Input/Output) takes milliseconds. This lack of support for high rate IO is exacerbated by modern multi-/many-core architectures whose inherent concurrency should intuitively yield higher aggregate throughput. In fact, many-core processors differ from more classic multi-core processors by offering simpler cores typically organized on a network-on-chip and running at a lower clock frequency than multi-cores [7], but suffering more from synchronization bottlenecks. This two paradigm shifts raise the problem of inadequacy between existing computer architectures and modern IO-bound applications.

A long trend of research has shown that bypassing the kernel, especially in the data plane fast path, can substantially improve performance [8, 22, 27, 41, 44]. This observation stems from the fact that context switches induce a substantial overhead on top of hardware capability [16, 39]. This led researchers to at least batch these costly context switches [5] and to sometimes exclude the kernel completely from the fast execution path. Such a user-level approach even managed to improve the performance of a web server running on Linux by an order of magnitude [27].

The drawback of user-level IO processing solutions is their lack of *protection*. They either separate IO processing threads from application threads that communicate through shared memory queues [22], or combine the IO event loop with the application event loop, in a *run-to-completion* (RTC) execution model. In both cases, they do not isolate the IO processing from the application execution. For example, the memory of user-level TCP/IP stacks is not mapped. This lack of constrained mapping does not prevent an application from corrupting the network through shared memory, hence affecting other applications. Some approaches enforce isolation in hardware to provide a coarse-grained protection but this does not prevent the application and the IO from sharing the same address space [34]. Hardware isolation does not provide a fine-grained protection based on policies, such as an access control list or a monitoring bandwidth limiter.

This observation even led to the common belief that user-level implementations could not provide high performance, low-latency and protection [5].

In this paper, we present the Distributed Library OS (DLibOS) to debunk the myth that high performance, low latency and protection cannot all be achieved at user-level. DLibOS achieves efficiency with zero-touch transmission and zero-copy reception without context switching. It offers protection by isolating the application from the IO tasks in different address spaces. In particular, its memory is initially mapped to prevent the applications from creating arbitrary packets or tampering with packet headers. DLibOS specializes isolated cores to different tasks similar to previous design proposals [45] but it differs in that it exploits hardware message passing for inter-core communication. This hardware message passing achieves a bandwidth two orders of magnitude higher than the one of shared memory [47]. This raises an interesting question regarding the peak performance one can achieve while distributing specialized cores.

To illustrate the benefit of DLibOS we developed a driver and a user-level unprivileged TCP/IP stack on a Tilera TILExtreme many-core machine. DLibOS does not interfere with the Linux kernel and allows to run concurrent networking stacks as long as each stack has at least one core and a unique MAC address. An interesting benefit of DLibOS is to offer timing guarantees by combining receive side scaling (RSS) with a shared nothing approach. For example, TCP acknowledgements are sent within a time bound even when the application is blocked, something that cannot be achieved with RTC. In summary, we drew the three following concluding observations regarding the advantages of distributed specialized cores for IO and application processing:

1. **Combining performance and protection does not require context switching.** DLibOS is precisely one example offering this combination. It separates the processes by the network-on-chip and registers the memory mappings in IO-TLB so that no applications can corrupt the network or other applications. Our experiments on the in-memory key-value store Memcached show more than 3 million requests per second (RPS) with a 99^{th} percentile latency at 500 μs, while all TCP solutions we know that evaluated Memcached without disabling its LRU policy, did not reach 2 million RPS [5, 16, 36].
2. **Distributing specialized cores does not increase tail latency.** Our implementation of another user-level TCP/IP stack using the run-to-completion (RTC) execution model reveals that DLibOS was more effective at reducing tail latency. This result may seem surprising given that RTC is known to achieve high data locality and limit the work and inter-core communication [27]. Interestingly, our profiling indicated that RTC tail latency is affected by (i) a lack of fairness between the network and the application to access CPU resources and (ii) a tendency to suffer from periods of serializations through contention.
3. **The flexibility offered by core specialization benefits concurrent applications.** On the one hand, our evaluation of a webserver application shows that a right balance of twice as many network-specialized cores as application-specialized cores boosts performance by 75% compared to the opposite balance of twice as many application cores as network cores. On the other hand, our comparison of webserver and Memcached key-value store confirms that one balance does not fit all applications as the performance of this key-value store peaks actually with twice as many application cores as network cores.

Our user-level network stack is not a panacea. First, it is complex in that the application must hold onto memory until it is acknowledged by remote cores while BSD sockets only need to hold on until memory is copied to socket buffers. Second, it requires the on-chip network, and, as we will explain, care needs to be taken to avoid deadlocks and stalling. Third, although the core-local resource usage minimizes cache conflicts/pressure on specialized cores by exploiting spatial locality, the RTC execution model exploits temporal locality that minimizes cache line eviction. Fourth, specializing cores may unnecessarily overprovision computing resources. Finally and as we present in this paper, RTC may perform slightly better than specialized cores, but without the same level of protection.

Roadmap. We start by presenting the background and motivations (§2). Then we present the design of DLibOS (§3), and describe its implementation (§4). Finally, we evaluate DLibOS on three benchmarks and two applications (§5), and present the related work (§6) before concluding (§7).

2 Background and Motivations

As extracting OS features exposed multicore applications to protection vulnerabilities through resource sharing, researchers have started investigating virtualization as an alternative source of performance to kernel bypass.

2.1 The Operating System Overheads

Performance incentivized innovations in Operating System (OS) architectures, sometimes as the expense of protection. On the one hand, the OS is the traditional way of ensuring *protection* by controlling the access of processes to resources. On the other hand, it has been known since the 90's that extracting features of OS into runtime libraries, so called *library OS* (libOS), could improve performance, in particular of concurrent applications [1]. LibOSes reduce the kernel support to its bare minimum, hence avoiding system calls in the common fast path. The exokernel OS [14] exploited the libOS to lower the level of abstractions of protected resources. It implements *zero-touch*, a mechanism that passes data directly to hardware without the processor touching them.

Some overheads are induced by the mismatch between the high performance of recent network devices and the original BSD socket interface. The model of TCP/IP stack initially designed by Berkeley in the 70's is still the predominant way to handle network communications, even though, in the meantime, the Internet has grown from a 13 nodes ARPAnet to a network of more than one billion hosts. The Linux networking stack relies on this BSD socket interface whose reads and writes require memory copies between user and kernel spaces. This interface, in particular the notification system (with `select`, `kqueue`, `poll` and `epoll`), requires system calls and context switches, which in turn introduces cache misses [39].

2.2 Scalability with the Number of Processing Cores

Similarly to the exokernel that modified the role of the kernel, Baumann et al. proposed the multikernel architecture and the corresponding Barrelfish OS for scalable multicore systems [4]. They consider a multicore processor as a distributed system, where cores communicate by exploiting cache line eviction for sending messages via the shared memory coherence. The satellite kernels and the corresponding Helios OS focuses on heterogeneous computer architectures and implement IPC through remote message passing [31].

A common technique to scale with the number of CPU cores, is to leverage *Receiver Side Scaling* (RSS) [29] that maps network packets to specific cores. In a recent proposal, Hruby et al. [20] present a design for network stacks that makes use of RSS to map packets to cores dedicated to run a TCP/IP stack in separate processes. They show that they can achieve comparable scalability to Linux. They exploit however a single network driver which would typically not scale on some manycore architectures. MegaPipe [16] provides a network I/O interface that offers asynchronous I/O communications between kernel and user space and batches context switches to limit their overhead. It also exploits RSS to assign each of the multiple *reception* (`Rx`) and *transmission* (`Tx`) queues to one CPU and its scalability is only evaluated up to 8 cores. FastSocket [26] is a kernel socket design that scales with the number of cores. It maintains the BSD-socket API and improves the NginX webserver throughput by 267% using the kernel.

To leverage multiple cores within the same microprocessor, several works have suggested to dedicate cores to various services [18, 19, 21, 37, 45]. In particular, FactoredOS [45] is an OS architecture tailored to scale on manycores. The idea is to distribute services across distinct cores to limit contention on TLB and cache resources and to expose a traditional system call interface for inter-core communication.

2.3 Unprotected User-Level I/O Processing

For communicating between multicore applications, a microkernel typically requires the application to context switch in order to deliver a message to the process responsible for handling the call, hence disrupting more the execution than in a monolythic system [19]. This is why, other solutions favored communication at user-level through the shared memory rather than with context switches, however, a malicious or buggy application can corrupt shared locations that the receiving process was already cleared to use [21]. Existing approaches using shared memory at user-level do not prevent a malicious application injecting arbitrary packets into the network.

Sandstorm [27] provides a user-space TCP/IP stack with *zero-copy*, the ability to receive and transmit network packets without copying data to memory. Sandstorm adopts a *run-to-completion* (RTC) execution model: letting a core execute both the network and application processing without experiencing preemption. Thanks to these two optimizations, Sandstorm presents a webserver that improves the throughput of the NginX[1] webserver with standard Linux by an order of magnitude. User-level RTC, however, can suffer from a lack of timing guarantees. As the application and the I/O processing must share the same processing core, the time taken by the application processing directly delays the I/O processing. For example, a networking stack could be unable to acknowledge data or handle timeouts due to the lack of preemption.

2.4 Is Virtualization Really Needed?

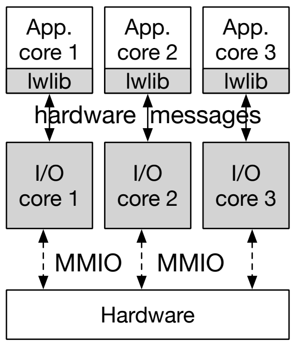
The lack of protection led some researchers to favor virtualization over user-level implementations. IX [5] achieves protection by placing the network processor in the Virtual Machine (VM) non-root ring 0, and the application processing in VM non-root ring 3. IX provides low latency and high throughput of network applications by adopting the RTC execution model. In contrast with user-level networking stacks that also execute RTC, IX benefits from interrupts to provide timing guarantees.

In general, virtualization is not trivial and often requires specialized hardware. The specialized hardware is necessary for IX to allow concurrent applications to benefit from the NIC. In particular, IX++ [9] tried to extend IX with *PCI Single Root I/O Virtualization* (SR-IOV), which allows a single hardware NIC to appear as multiple virtual NICs. Unfortunately, their result was limited by the Intel 82599 NIC chipset and could not exploit RSS, a limitation that could be addressed with the recent Intel xl710 NIC. Due to this limitation, IX++ can only use a single CPU core.

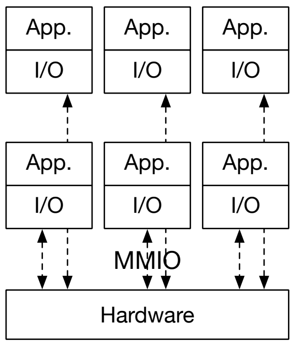
The OpenOnload project[2] exploits an elaborate virtualization mechanism to offer a secure user-level networking

[1] https://nginx.org/en/.

[2] http://www.openonload.org/.



(a) DLibOS (shaded area) is distributed on specialized cores that communicate through hardware message passing and MMIO



(b) Traditional user-level libraries run-to-completion on a single core communicating by shared memory or context switches

Figure 1. DLibOS exploits hardware message passing to avoid context switching and enforce isolation

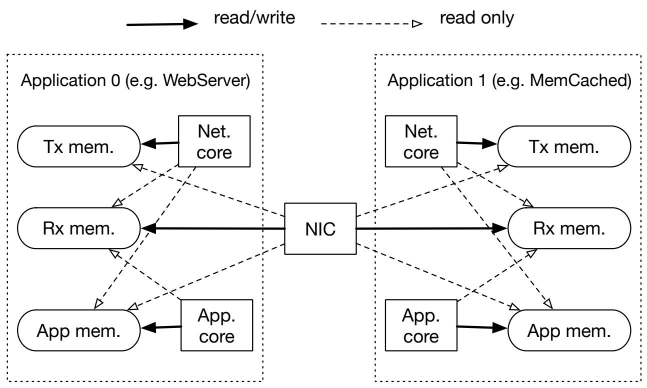


Figure 2. The partitioning of the memory prevents applications from corrupting network and inter-application corruptions

stack with the standard BSD socket interface. Similarly, its virtualization mechanism requires the dedicated hardware of the SolarFlare NICs.

More recent approaches took this idea of virtualization further. Arrakis [34] extends the Barrelfish OS by integrating SR-IOV to obtain a significant performance gain and adapted it not only to networking I/O but also storage I/O. EbbRT [36] exploits the run-to-completion model and removes almost completely the kernel abstraction by running within a virtual Xen environment where the kernel is replaced by a library runtime.

3 Design

In this section, we present the design of DLibOS that revisits library OSes to provide protection at user-level. The four key principles are its core specialization, its memory partitioning, its message-based asynchronous interface and its flow pinning strategy.

3.1 Distributed and Specialized Processing Cores

The prominent design of DLibOS is the specialization of its CPU cores and the isolation of their respective processes communicating through hardware messages. The rationale of this specialization resides in offering reliability by isolating processing tasks from one another and flexiblity in the provisioning of processing resources. For example, we will show in §5 that an in-memory key-value store application needs to dedicate more cores to the application processing whereas a webserver needs to dedicate more cores to the IO processing. Similar specializations have already been envisioned to leverage multicores and manycores machines [18, 19, 21, 37, 45], however, they require either context switches or coherent shared memory accesses to implement a so-called "software message passing".

The novelty of our specialization is its distribution: cores communicate with "hardware message passing". Not only does hardware messaging offer efficient inter-core communication without costly context switches, but it also allows us to implement protection as we will explain in §4.5 by spawning multiple processes with registered memory mappings. This core design is depicted in Figure 1 where DLibOS appears on the left-hand side (Figure 1(a)) and the legacy run-to-completion alternative appears on the right-hand side (Figure 1(b)). For the sake of simplicity, Figure 1(a) depicts 6 balanced cores with half of the cores configured to process IO and the other half configured to process the application functions. Figure 1(b) depicts a generic RTC architecture, similar to existing ones [27], where IO management and application functions run sequentially and therefore share the same memory address.

3.2 Memory Partitioning and Zero-Copy Accesses

To protect the application while enabling high throughput and low latency, DLibOS partitions the memory into three distinct regions: the transmission (Tx) memory, the reception (Rx) memory, and the application (App.) memory while ensuring zero-copy or that neither Rx nor Tx copy anything to memory.

This original partitioning of the memory creates one address space for the application layer, another one for the IO handling layer and a last one for the hardware/network component. Then, we propose to offer differentiated read/write rights to these various address spaces depending on the function accessing them. For example, only the application (resp. IO or hardware) can both read and write on the application (resp. IO or hardware) address space whereas the other layers can only read this space. As an illustration, we present in Figure 2 how, within an application, one application core

	events	parameters	description
net ⟶ app	new_conn	$id, listen\_id$	a connection (cxn) is opened
	new_data	$id, \{ptr(\mathit{offset}), len\}$	an array of msgs is received
	data_acked	$id, count\_acked$	data has been acknowledged
	remote_closed	id	cxn was closed remotely
	cleanup	id	cxn has been closed and all resources can be safely freed

	commands	parameters	description
app ⟶ net	accept	$id, listen\_id$	accepts a connection (cnx)
	connect	*cookie*, *dst_ip*, *dst_port*	opens a cnx
	recv_done	*id*, *count_received*	credits rx window, releases rx mem.
	send	$id, \{ptr(\mathit{offset}), len, \mathit{flags}\}$	sends a scatter/gather array of data
	shutdown	*id*, *flags*	closes connection
	close	id	closes and frees resources of a cxn

Table 1. Hardware messages for inter-core communications

shares a single address space where the network (IO in this case) core accesses another one.

In addition to this intra-application memory protection, DLibOS guarantees that inter-application memory access is forbidden for any application cores. This property is illustrated in Figure 2, where none of the application cores from Application 0 (resp. Application 1) can either read or write memory address spaces defined within Application 1 (resp. Application 0).

Overall, we leverage this partitioning to provide zero-copy for reception: the data is passed by reference through messages between the NIC and the application without any copy. As we will explain in §4.6, our design allows for further optimization to achieve zero-touch on the `Tx` path so that the CPU does not even touch in-memory data.

3.3 Asynchronous API without System Calls

In order to maintain low latency and achieve a greater IO efficiency we introduce a fully asynchronous communication system between the application and the IO processing. This interface comprises a series of events and commands described in Table 1. Each function of this interface is invoked through the sending and reception of hardware messages between cores. The asynchrony stems from a "fire-and-forget" invocation principle up and down stream. This asynchronous interface is thus also non-blocking and similar to the "notification completion" of megaPipe [16], however, this "notification completion" emulates software message passing through shared memory.

This asynchrony may appear detrimental at first glance, as it makes it harder to distinguish between a delay and a failure. In our case, however, we rely on a loss-free message passing guaranteeing that no messages can be dropped: the implementation of inter-core communication is detailed later in §4.3. In the context of application to network interaction, this asynchrony constitutes the biggest difference between our DLibOS and legacy architecture relying on Berkeley sockets. Indeed, in the legacy model, IO interaction with the network can be either synchronous and blocking, or synchronous and nonblocking, whereas DLibOS is asynchronous and non blocking.

3.4 Chip Locality and Balanced Flow Pinning

One of the final key design element in DLibOS relates to the use of many-core architecture coupled with Network-on-Chip inter-core communications. In order to fully use this powerful architecture, core specialization needs to be selected carefully to both limit communication relay by cores and maximize memory localization (placement of data). Limitations of inter-core communication between IO handling and applications is done in DLibOS by carefully placing the different specialized core on the Network on Chip grid. We explain how we optimize this placement in the particular case of our hardware in §4.4.

We do flow pinning to distinct cores in two different steps. First, we ensure that a flow is processed by the same network core using the flow hashing of RSS. Second, we ensure that the flow is processed by the same application core by statically pinning the flow to an application core via a pre-configured policy such as round robin. More precisely, the network core load-balances the connection events by dispatching `new_conn` events to applications cores. Any application core is allowed to accept the connection by sending a response to the issuing network core. Once an application core accepts the connection it is statically pinned to the corresponding flow and becomes responsible of treating all its packets. Each network core maintains a list of the application cores it is allowed to communicate with based on this pinning.

4 Implementation

Our implementation runs on top of Linux on the TileGx-36 processor. We implemented a userspace TCP/IP stack and userspace driver on top of Tilera mPIPE API [11]. We use RSS flow hashing to distribute packets to cores, allowing us to avoid sharing and synchronizing states between network cores. Our driver makes use of bounded batching when interacting with hardware `Rx` and `Tx` queues, as well as the generation of new packets. The network stack and driver are designed with zero-copy/touch in mind, using scatter-gather IO and checksum offload to avoid polluting caches. We implemented a "socket" abstraction layer that exposes events and commands to the application, this layer is able to function in either DLibOS or run-to-completion mode, without changing the application itself.

4.1 Run-to-Completion as the Baseline

For the sake of evaluating DLibOS, we also implemented a traditional RTC user-level TCP/IP stack on Tilera as a baseline. To this end, we defined a "socket" abstraction layer exposed to the application that is implemented with both

L1$ hit	L2$ hit	L3 (distributed) $ hit	UDN latency	UDN bandwidth	shared memory bandwidth
2 cycles	11 cycles	40 cycles	1 cycle per hop	60 Tbps	170 Gbps

Table 2. The messaging latency is one order of magnitude lower than cache ($) latency.

DLibOS (Figure 1(a)) and RTC (Figure 1(b)). In RTC mode, the application runs in the same address space as the network stack and the driver. The network stack invokes the application event handler callback upon new events instead of dispatching messages, similarly the socket layer directly calls relevant stack functions instead of dispatching commands on the network-on-chip. As RTC does considerably less work than DLibOS, one could expect RTC to perform better than DLibOS at the cost of isolation and timing guarantees, however, we will explain why this is not always the case depending on the application in §5.

4.2 Shared Nothing on Name Spaces and NIC Queues

For our implementation to scale, we adopted a shared nothing architecture for DlibOS datastructures. We adopt a single writer principle for IO data as explained earlier in Figure 2. As originally stated, the shared nothing architecture consists of preventing processors from sharing memory or storage [40]. In the context of multicore, this architecture helps scaling to a large number of processing cores by reducing bottleneck effects induced by the synchronization on shared resources.

To avoid synchronization between network cores and application cores, our implementation features multiple file descriptor name spaces: one per network core, hence resolving the shared namespace issue [10]. As a result, we have multiple file descriptors per application process. In particular, we identify a file with a couple $\langle \textit{net-core}, \textit{id} \rangle$, this further avoids synchronization. By contrast, traditional approaches have one name space per process. For example, POSIX needs an integer for a file identifier that must be synchronized among all threads.

To avoid synchronization between the NIC and the networking cores, we use flow hashing based on RSS. In DLibOS, each networking core serves a single NIC queue, eliminating the need for synchronization among multiple network stack cores. Flow hashing is a common technique that recently proved promising at increasing performance of short-lived connections in a kernel socket design [26].

4.3 Network-on-Chip Messaging and Routing

Our implementation uses the User Defined Network (UDN) on the TileGx-36 processor for intercore messaging rather than writing messages into shared memory. On Tilera, the messages are single-cycle dispatched so that the core can continue processing while the message continues being dispatched.[3]

[3]https://lwn.net/Articles/500232/.

We use the on-chip network for commands and events between DlibOS and applications, relying on shared memory for IO data. Metadata requires latency sensitive point to point messaging and is better suited toward the on-chip network. The lower volume of metadata exerts less pressure on the on-chip network buffers and congestion. The on-chip network also provides a degree of protection, presenting an append only queue without requiring context switching.

Using cache eviction for communication as used in Barrelfish [4] can lead to poor performance on some architectures [42] or may be impossible on others [17]. In particular, on Tilera only the L3 cache is shared among the same chip and a L3 cache hit takes 40 cycles. As the cache communication is pull-based, the sending core writes in a local cache and the remote core has to read, which increases further the cache-to-cache communication latency. As a result, the hardware messaging latency (UDN latency) is one order magnitude lower than the L3 cache latency as summarized in Table 2. In addition, the on-chip network offers significantly larger throughput than shared memory, however, care must be taken to avoid deadlocks due to the nature of this network.

The on-chip network leverages *wormhole* routing: the header of the message reserves a route, hence preventing interleaving of different messages along this route. The network does not drop messages, each core has a non blocking switch with send and receive buffers to handle bursts of messages. If a message in the receive buffers is not consumed, then messages may back up through the network. In addition, attempting to write messages to a full buffer will stall the core until the message is written. As a result, the limited message buffer space can result in deadlock if pairs of cores send messages without ever draining them by reading from receive buffers.

DLibOS achieves deadlock-freedom by credit check similar to NoCMsg [47]. More precisely, we check the hardware register that stores the FIFO credit of the core local switch and avoid sending if this would stall the core processor pipeline. When we are unable to send messages, we queue events/commands in memory but continue to process incoming messages to ensure overall progress, we then periodically attempt to retry sending messages. We also use a highly compressed message format, and allow batching of some messages to reduce pressure on network-on-chip buffers in order to avoid having to resort to retry sending messages.

4.4 Core Layout and Quadrant Configurations

We structure the layout of cores to minimize the number of contended paths on the network. The wormhole routing

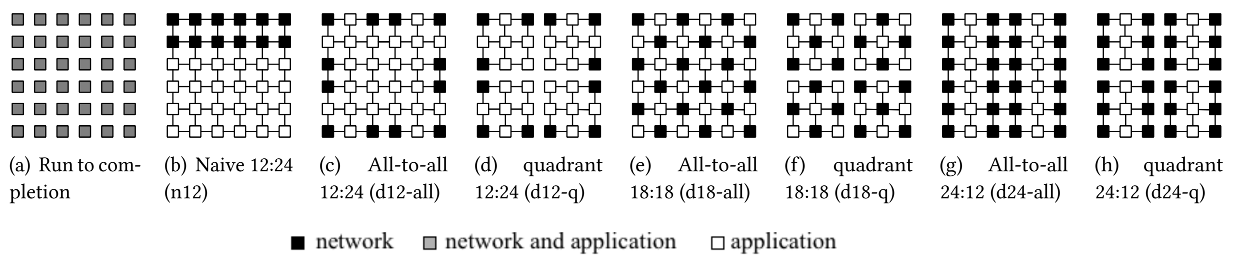

(a) Run to completion
(b) Naive 12:24 (n12)
(c) All-to-all 12:24 (d12-all)
(d) quadrant 12:24 (d12-q)
(e) All-to-all 18:18 (d18-all)
(f) quadrant 18:18 (d18-q)
(g) All-to-all 24:12 (d24-all)
(h) quadrant 24:12 (d24-q)
■ network ▨ network and application □ application

Figure 3. Core pinning layout and inter-core communication with ratio net:app cores

can result in path contention as other messages are blocked while a message traverses the shared path before closing the wormhole (cf. §4.3).

DLibOS specializes a subset of cores for application and another subset for networking with specific layouts as opposed to a RTC layout (Figure 3(a)) where each core processes both application and networking. A first specialized layout consists of naively specializing the first cores for the networking and specializing the remaining ones for the application as depicted in Figure 3(b). A second specialized layout consists of partitioning the TileGx-36 processor into 4 quadrants of 9 cores each, as depicted in Figures 3(c)–3(h). We can either restrict inter-core communication within a quadrant, hence called "dedicated-quadrant" (Figures 3(d), 3(f) and 3(h)) or allow communication across quadrants, hence called "all-to-all" (Figures 3(c), 3(e) and 3(g)). Restricting communication within a quadrant results in shorter paths; it reduces the average message latency, the number of contended hops and cross chip traffic. The drawback is to lead to poorer utilization when the quadrants are heterogeneously loaded.

4.5 Protection Through Separate Address Spaces

While regular user-level network stacks have to trust the application for correct network behavior, DLibOS does not assume that the application is trusted. For example, running a user-level networking stack such as mTCP [22] on separate threads can still result in corruption of the networking stack as both the application and network run within the same process.

By contrast, DLibOS runs an application in a separate process from the network stack but registers shared memory mappings and shared accesses to the network-on-chip. Each of the memory mappings is registered into the NICs IO-TLB in order to translate virtual addresses and provide isolation. Only the memory registered with the IO-TLB can be transmitted, and the arbitrary private application memory (cf. §3.2) such as keys cannot be transmitted unless explicitly copied to this memory. On the reception side, the applications can only access the Rx memory, which is mapped read-only to the application such that it cannot interfere with the operation of the networking stack. On the transmission side, the application is required not to modify memory until the data has been acknowledged, however failure to do this will only result in incorrect data payloads.

By isolating processing cores, DLibOS also provides some level of reliability. More specifically, if a process crashes, then we can often simply restart it without affecting concurrently running applications accessing the same NIC or the OS itself. As a result of this inherent protection, DLibOS can also be used to enforce security policies such as firewalling, rate-limiting or even prioritizing certain types of traffic. DLibOS exposes a higher level abstraction to the application, which can allow the transparent implementation of proxying, tunneling and encryption without the knowledge or cooperation of the application.

Finally, it is interesting to note that isolating functionalities on specialized cores rather than letting each core run to completion also presents some advantages in terms of real-time guarantees. For example, TCP acknowledgment is decoupled from the application function as opposed to a run to completion.

4.6 Zero-Touch Transmission & Zero-Copy Reception

Reception. Initially, the memory registered for reception with the NIC is mapped read-only to the application and network cores. During the reception, the networking stack maintains a list of credits for each application core and drops packets bound to that core if the application has not returned enough buffers. The reception (Rx) of a packet follows these steps in chronological order:

Rx1. The NIC receives a packet. It hashes the tuple ⟨ *senderIP*, *senderPort*, *receivedIP*, *receiverPort*, *protocol* ⟩ to find the corresponding network core before the NIC fills the Rx memory. It updates the Rx ring by writing a buffer descriptor pointing to the associated Rx memory.

Rx2. The network core keeps polling its Rx ring buffer and advances the tail pointer when consuming it. It notifies the application by sending it a pointer to the payload, through hardware message passing. If the data corresponds to a received TCP ACK, the network core sends an acknowledgement to the application to free the application memory located where the acknowledged data reside.

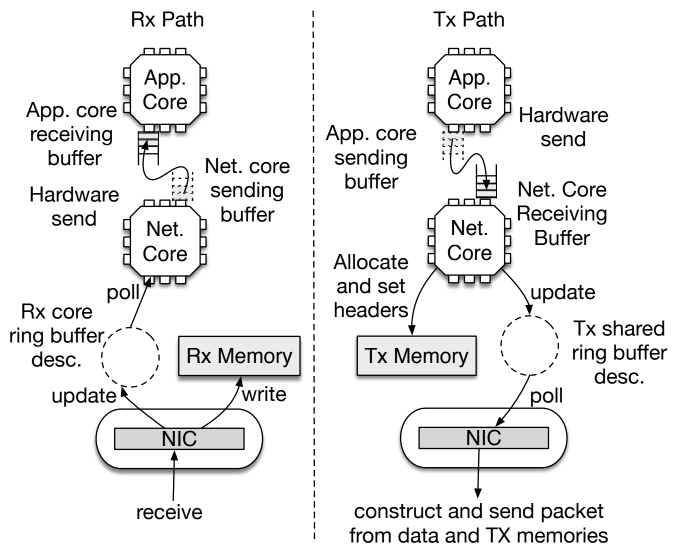


Figure 4. The reception (left) and transmission (right)

Rx3. The application receives the pointer. Once the application is done processing the data, it sends an acknowledgment through hardware message passing notifying the network core to free the corresponding `Rx` memory locations.

We mapped `Rx` buffers at the NIC reflecting a bimodal packet distribution [30], with 5000 small (256 bytes) and 2500 large (1664 bytes) buffers.

Transmission. Packets are constructed using scatter gather IO, where headers are allocated in transmission (`Tx`) memory inaccessible to the application, and application memory can only be referenced in packet payloads. First, the packet headers are immediately freed upon Direct Memory Access (DMA) into a stack so that they can be re-used for improved temporal cache locality, this also significantly reduces the memory requirements. We found 30-40 `Tx` buffers per core to be more than sufficient for our requirements due to the short lifetime of `Tx` buffers. Second, the packet payloads can consist of multiple regions of application memory where large regions can be split across multiple packets.

This approach of using scatter gather IO avoids exposing the underlying segment size to the application and allows the same application buffer to be sent across multiple connections simultaneously. Sandstorm and Exokernel [27] relied on pre-segmented data with pre-calculated checksums, and required multiple copies of application data in order to serve concurrent requests for the same data. We use checksum offload to avoid pre-calculating checksums and scatter gather IO to avoid multiple copies of application data. This design allows certain applications to transmit zero-touch, as the response memory never touches the CPU, hence reducing cache pollution. An application can benefit from this design by pre-storing responses within the application memory and passing references to the networking cores.

The transmission (`Tx`) of a packet consists of the following steps in chronological order:

Tx1. The application builds a message in the send buffer of the application core and sends this message through hardware message passing to the network core that receives it in its reception buffer.

Tx2. The network core looks up the connection and queues the message on the associated connection. It then combines the data into TCP segments, writes the network header and puts the descriptor of the TCP segments in the `Tx` memory. The network core then writes in the `Tx` memory the header and a pointer to the payload of the appropriate segment with a special flag to indicate the last segment for the packet.

Tx3. The NIC polls from the `Tx` ring to find the headers and increment the tail pointer.

We use a hierachical timing wheel for implementing TCP timeouts [43].

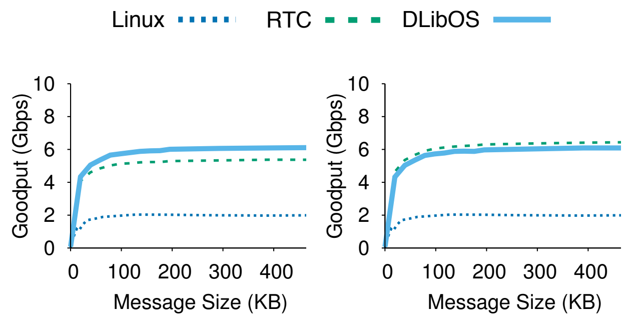
5 Evaluation

In this section we compare the performance of DLibOS to vanilla Linux and our RTC implementation. To this end, we ran (i) a port of the Memcached caching system, (ii) a port of the NGINX webserver application and (iii) a novel webserver application. We evaluate the performance through the benchmarks `NetPIPE`, `mutilate` and `wrk`, and Facebook's key-value cache workload.

5.1 Experimental Settings

We implemented DLibOS and RTC on one socket of a Tilera TILExtreme machine running Linux 3.10.90 and featuring a total of 4 TileGx-36 sockets. Each socket embeds 36 cores running at 1.2 GHz, organized in a mesh and interconnected with a network-on-chip. Each socket has also 4 Ethernet ports with 10 Gbps capability each, interfaced with the Tilera proprietary mPIPE, that we bonded at the switch using an L3+L4 hash.

For stress testing the Tilera server, we generated a client load from 8 identical machines, each equipped with an 8-core hyperthreaded Intel Xeon Processor E5-2620 v4 running at 2.10 GHz with 20 MB cache and 64 GB of memory, connected to the DLibOS via a 10 Gbps SFP+ cable through an Intel x710 NIC. The monitoring of Memcached performance was performed by a dedicated machine with a dual 8-core hyperthreaded Intel Xeon E5-2450 processors running at 2.1 GHz for a total of 32 hardware contexts and 64 GB of memory and connected with a dual 10 GbE ports on an Intel x520-DA2. All clients generated traffic goes through a Mellanox MSX1016X-2BFR switch with 64 SFP+ 10 GbE ports except in the case of the `NetPIPE` benchmark where we connected



(a) NetPIPE performance (by default, the server copies the payload of each message it receives)

(b) Performance when the server copies only the payload of the first message

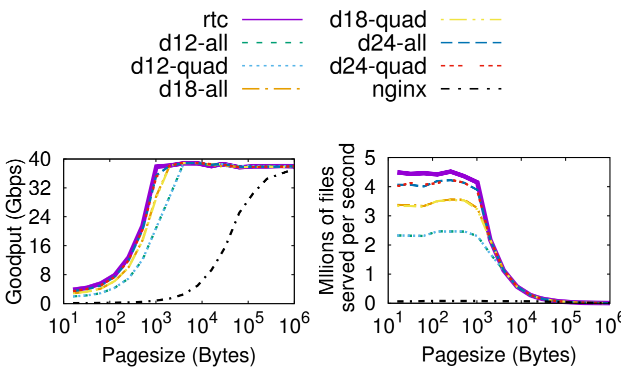
Figure 5. NetPIPE performance with DLibOS, RTC and vanilla Linux for varying message sizes and system software configurations with *d* followed by the number of network cores (all the remaining cores are used by the application)

two of the TILExtreme sockets with a direct cable to bypass the switch overhead.

5.2 Performance: High Throughput and Low Latency

We first ran NetPIPE, a popular micro-benchmark of a ping-pong protocol [38]. In this benchmark, a client and a server open a connection and exchange messages of a given fixed size to calibrate the latency and bandwidth of a single flow. The classic NetPIPE benchmark is called "copy mod": every time the server receives a new message, it copies it to the Tx buffer and resends it back to the client. Then, the client waits until the message is entirely received, validates that it matches what was sent and restarts. To emphasize the overhead of the distributed specialized core design of DLibOS, we extended NetPIPE with "nocopy mod" where the exchanged messages are assumed to be the same and the server copies only the first message of the series. In this case, the server waits until each message is received, discards it and sends the buffer copied from the first message back to the client.

Figure 5 compares the *goodput* of DLibOS, RTC and vanilla Linux under NetPIPE as the number of messages per second exchanged time the size of the messages. In particular, Figure 5(a) depicts the performance in "copy mod" whereas Figure 5(b) presents the performance in "nocopy mod". DLibOS outperforms RTC and vanilla Linux by up to 13% and 200% respectively in the standard NetPIPE benchmark. In our extended "nocopy mod", the application does minimal work and we can observe the overhead of specialized cores, where RTC outperforms our DLibOS by 6%. In "copy mod", DLibOS benefits from parallelism within a connection by having the application perform the copy on a separate core. These results can be explained by the CPU limitations in Tilera sockets: each core has frequency of only 1.2 Ghz, hence the



(a) Goodput vs. File size

(b) File per seconds vs. File size

Figure 6. DLibOS and RTC webserver performance compared to NGINX using various layout configurations and different file sizes.

limited goodput. We also observed that for 64-byte messages and for the classic NETPIPE, the one-way latency of DLibOS and RTC is 6 μs, whereas the one-way latency of Linux is 31 μs, confirming that DLibOS achieves low latency.

5.3 Stressing the Network: Webserver

We implemented a webserver application that leverages DLibOS and RTC by distributing TCP connections to multiple cores and responding to HTTP client requests. The server acts in-memory and serves static data from webpages preloaded into contiguous memory. We compare our results to two other scenarios. First, we evaluate the standard NGINX webserver running on TILExtreme configured to serve file from a ram-disk, with disabled logging, disabled compression and increased listen backlog. Second, we evaluate our own webserver running on our RTC user-level TCP/IP stack.

The* wrk *benchmark. We set the wrk benchmark to open 2048 connections spread over the 8 client machines, each with 256 connections, and varying the requested file size. Similar to the Sandstorm benchmark setting, each HTTP client generates a series of requests, with a new application connection being initiated immediately after the previous one terminates, whereas we enabled keepalives at HTTP level. We do not exploit HTTP pipelining to maximize throughput, each connection has only 1 outstanding request at a given time. Figure 6 compares the performance (goodput in Gbps and files server per second) of NGINX and our webserver running with RTC, and DLibOS under different layout configurations.

Figure 6(a) shows that our DLibOS and RTC exploit most of the 40 Gbps capability of the TILExtreme socket. In particular, NGINX cannot serve small HTTP object (< 2 KB) as fast as our user-level TCP/IP stacks, as it requires MB object sizes to perform close to line rate. To illustrate the causes of DLibOS performance advantages, we varied the ratio of networking vs. application cores in some of the layout configurations previously listed in Figure 3. Figure 6(b) shows clearly that the number of files served per second does not significantly depend on the inter-core communication pattern but is actually closely related to this ratio. In particular, a ratio of net-to-app of 2-to-1 improves the performance of a ratio 1-to-2 by more than 70% for file sizes less than 1KB. Majorities of I/O cores proved suitable on other lightweight many-cores apps [15]. Above 1KB, DLibOS and RTC are limited by line rate (note the logscale on the x-axis that hides a drop in the number of files served that is linear in their size growth). This skewed ratio is explained by the fact that the webserver stresses more the network than the application.

5.4 Stressing the System: Memcached

To complement the webserver application results that stressed the network, in this section we evaluate performance under Memcached, an in-memory key-value store built on top of the libevent framework that is deployed in industry [32] to provide a high-throughput and low-latency caching for persistent servers.[4] In the following evaluations, we use all the available 36 cores.

The* `mutilate` *benchmark with the Facebook workload. We have evaluated the performance of Memcached using the `mutilate` benchmarking client [24]. This benchmark framework aims at generating a targeted Requests-per-Second (RPS) across several and coordinated clients while an additional client measures the latency without the heavy load of the request. In our configuration we used the eight homogeneous Intel Xeon E5-2620 v4 to generate the desired load while the dual-processors Intel Xeon E5-2450 sampled the latency. Overall, we configured `mutilate` to create a total of 1392 client connections, where we randomly sampled the latency from 32 of these connections on the observer computer.

We also configured the `mutilate` benchmark to generate the user-account status information (USR) workload of the Facebook trace [2] in order to stress the system with a high packet rate. This workload was used in recent evaluations of libOSes [5, 36] and is dominated by 99% of GET requests with short keys (< 20 bytes), 2 B values and minimum sized TCP packets.

Memcached-specific settings. In our testing environment, clients were configured with a pipeline depth of four requests per connection to keep-up with the configured RPS. On the server side, we ran Memcached in all configurations (RTC, and DLibOS) with 1 million records. We further increased the size of the hashtable with the `hashpower=20` command line option, and used pre-allocated slab memory.

[4]http://memcached.org/.

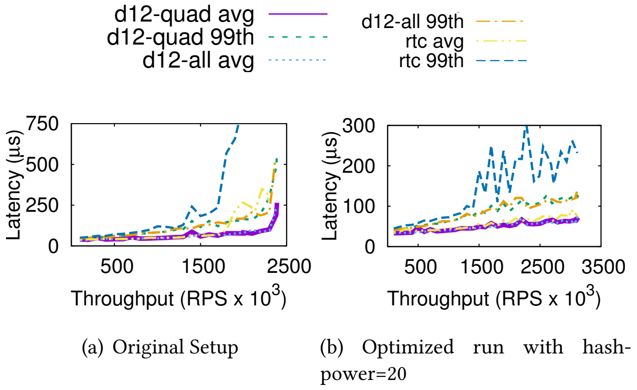


Figure 7. Memcached performance comparison with DLibOS and RTC under various requests per second.

In order to support Memcached we replaced the libevent loop with our own event loop, in the RTC configuration this was the network processing event loop, in the DLibOS configuration this polled the network-on-chip.

We modified Memcached to support zero-copy operation, since most requests are small they are most likely to not span multiple TCP segments and would benefit from zero-copy. On the receive side we modified the read function to return contiguous requests, if a request spanned multiple segments it would be copied to another buffer. On the transmit side we modified Memcached to support multiple outstanding responses that have been sent but have not been acknowledged by TCP. The Memcached slab allocator was modified to use memory registered for transmission with the NIC. We prevent freeing any Memcached items that still have outstanding acknowledgements by adding a second reference count to Memcached items, the original reference count is used only for unlinking from the hashtable. We intentionally left the remaining code intact.

Solution	Config.	Avg. latency (μs)	Unl. 99^{th} latency (μs)	RPS@500μs 99^{th} %-tile	RPS@5000μs 99^{th} %-tile
Linux		405.9	846.2	N/A	100K
RTC		35.8	44.9	3.1M	-
DLibOS	all	32.8	40.9	3.1M	-
DLibOS	quad	33.0	40.9	3.1M	-

Table 3. Comparison of Memcached performance

We also run Memcached on the Linux kernel but we were unable to achieve the service-level agreement (SLA) of 500 μs under the normal load (hence "N/A"). Results are depicted separately in Table 3.

Memcached throughput with DLibOS. Figure 7 presents the throughput-latency results for two Memcached configurations, namely with and without the hashtable optimization. Figure 7(a) illustrates that DLibOS, in the original setup

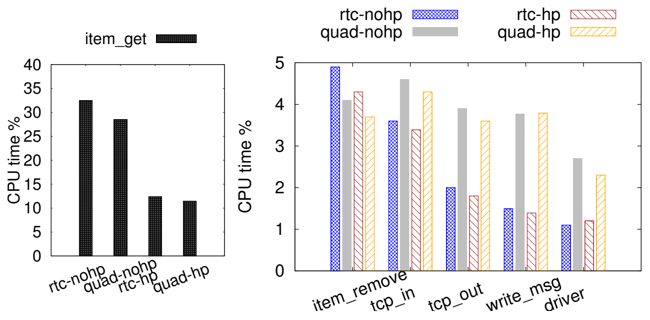


Figure 8. Profiling memcached at maximum RPS with DLibOS and RTC.

of Memcached, can reach an unprecedented 2.4 million request rate while maintaining an SLA of 500 μs measured at the 99th percentile. This result corresponds to an improvement of nearly 40% compared to the RTC deployment, which reached a maximum request rate of 1.8 million RPS.

When increasing the size of the hashtable with the `hashpower=20` option, Figure 7(b) depicts that DLibOS achieves a maximum request rate of 3.1 million RPS (even without going over the 500 μs SLA measured at the 99th percentile). In Figure 7(b), we further observe that, in this configuration, DLibOS obtains a less versatile 99th percentile latency than the RTC version. However, we were not able to overload our server due to a lack of clients systems.

Memcached profiling with DLibOS. In order to understand the performance of DLibOS with Memcached, we present in Figure 8 the profiling of our Memcached implementation that is obtained through sampling. We can see that, in all cases, DLibOS better utilizes the CPU than RTC. In particular, in the default setting, we can see that Memcached spends 32.5% and 28.5% of its CPU time in the `item_get` function, respectively for RTC and DLibOS. This function is responsible for accessing the item in memory via a striped lock table of size 2^h entries. Hence, increasing the size of this lock table with the hashpower option h reduces lock contention. The fact that in Figure 7, we see that the average latency for RTC is significantly closer to DLibOS when the hashpower option h is large seems to indicate that RTC suffers more from contention than DLibOS.

hashpower	14	16	18	20
DlibOS / rtc	0.71	0.81	0.77	0.80

Table 4. Ratio of cpu samples spent acquiring locks for different Memcached hashpowers

To further validate our hypothesis, we did additional profiling. Table 4 indicates that DLibOS spends less time acquiring locks than RTC in all cases and RTC is impacted by a low hashpower of 14 to a greater extent than DLibOS. From this we can conclude that an RTC workload is impacted by increased lock contention. In particular, this lock contention is further impacting the service latency, which was captured in the difference in tail latency (99th percentile) in Figure 7 between RTC and DLibOS. It is noteworthy that an RTC configuration is generally impacted by greater lock contention by having a greater number of cores (50%) contending on the striped lock table.

DLibOS spends more time in network and driver related functions compared with RTC both with and without the hashtable modification, we believe that this is due to a lack of fairness in RTC where both the application and the network are competing for CPU time without pre-emptive scheduling. This lack of fairness could be contributing to the difference in tail latency even when the hashtable modification is in place.

We also profiled Memcached running on the Linux kernel in order to understand why it was unable to meet the given SLA. We observed 22.77% of CPU time was spent in kernel spin locks, the combination of lower frequency cores and higher core counts exacerbates this locking overhead. There are probably other causes but enumerating all of them is out of the scope of this paper.

Reflexion on memcached performances. Overall, through the evaluation of DLibOS with Memcached, we have demonstrated that not only DLibOS could reach unprecedented rates of RPS under the 500 μs SLA but that it outperformed RTC mode. This achievement can be explained by a better serialization of key Memcached functions hence reducing the number of contentions. Through this result, we show that both user-level architecture can achieve very high performance and low latency.

6 Related Work

Handling incoming packets with commodity computers raises several OS challenges. An alternative is to bypass the network stack entirely or partially. Systems such as mTCP [22] and Sandstorm [27] propose a fully functional TCP/IP stack down to the network, at user-level. While appealing for performance, they complicate the resource sharing by depending on hardware. For packet IO, mTCP relies on DPDK whereas Sandstorm relies on Netmap [35]. Besides mTCP, other frameworks, such as NetBricks [33] or ClickOS [28], reside on top of DPDK user-space. They allow end-users to easily implement network processing functions with zero-copy. These frameworks focus, however, on packet processing putting aside the complexity of the TCP state machine. At the frontier between packet processing and TCP solutions, some approaches [16, 46] provide a network IO interface that optimizes the communication between kernel and user space and batches context switches to limit their overhead. MegaPipe [16] offers protection while StackMap [46] favors performance without preventing direct buffer accesses.

	Performance	Communication	Context switches	Isolates NW stack	Flexibility, Extensibility
Vanilla kernel	N	syscalls	Y	Y	Y
mTCP [22]	Y	function calls	N	N	Y
Sandstorm [27]	Y	function calls	N	N	Y
IX [5]	Y	(batched) syscalls	Y	Y	Y
HW [23, 25]	Y	-	-	N	N
DlibOS	Y	HW msgs	N	Y	Y

Table 5. Comparison of DlibOS and existing solutions

An exokernel consists of externalizing an abstraction to user-space, leaving the protection to the kernel. It typically exposes hardware securely, physical names, allocation and revocation. The Exokernel [12, 14] provides a minimal kernel or a derived fast path to bypass the OS functions. Barrelfish [4, 31] also advocates an externalization of hardware abstraction to user-space but to scale to modern multicore machines. Building upon exokernels are libOSes that offer abstraction in user-space but give the possibility of tailoring the abstraction to the application. IX [5] efficiently leverages the networking stack and designs a more efficient socket API than the BSD interface. It builds upon Linux and adopts an RTC execution mode to maximize data cache locality by scheduling application threads appropriately with the delivery of batches of events. Arrakis [34] also handles storage IO and maps the virtual NIC to applications with SR-IOV and the IO memory management unit (IOMMU). It separates the control plane in kernel space from the data plane in user space but builds upon the Barrelfish OS. Since the control plane is not on the critical path, the packets are sent and received directly in user-space. Finally, EbbRT [36] is a LibOS that proposes run-to-completion in a virtualized environment where the application shares a runtime with the networking stack.

FactoredOS [45] is an architecture proposal where cores are specialized into services, like file system or network processing. This architecture is motivated by the evaluation of performance bottlenecks in existing OSes and the simulation of different cache sizes to measure the cache miss rate on the Linux OS, but as far as we know there is no implementation of this proposal. IsoStack [37] runs on a single specialized core to minimize data sharing, however, its networking stack is sequential. More recently, the NEaT scalable network stack [20] was proposed to partition the stack across isolated process replicas handling independent requests. The aforementioned solutions explored functionally partitioning different stages of network processing onto separate cores but rely on software message queues or context switches.

In contrast with software-based solutions, several approaches, like RDMA [13], NetFPGA [6], Intel flow director [25] were implemented or suggested [23] to provide hardware accelerated key-value stores. These hardware accelerations lack flexibility and rely on vendor support for updates. In a similar vein, MICA [25] offers a specialized key-value store that executes run-to-completion to treat exclusively UDP datagrams on top of DPDK.

Table 5 presents a comparison overview of DLibOS against the closest related work.

7 Concluding Remarks

We presented DLibOS that demonstrates the possibility of combining high throughput, low latency and protection at user-level. To this end, it combines for the first time specialized processes with hardware messaging. Its evaluation in a user-level TCP/IP stack against a traditional run-to-completion user-level stack indicates better tail latency and revealed the importance of balancing the specialization ratio.

Future work includes several directions. First, while the DLibOS implementation does not support storage, this could be added with NVMe SSDs and Intel SPDK[5] to run other types of applications like persistent databases. Second, our current implementation relies on TileGx but we intend to port DLibOS on other architectures. Third, DLibOS does not support dynamically spinning I/O cores up and down to satisfy a given SLA while minimizing idle resource utilization, this could be achieved by modifying the messaging protocol and the driver to support flow migrations. Finally, while multiple applications can benefit from multiple DLibOS instances on the same machine, we intend to extend the memory partitioning to support running multiple applications from a single DLibOS instance to reduce the number of cores needed.

Acknowledgments

This research was supported under Australian Research Council's Discovery Projects funding scheme (project number 160104801) entitled "Data Structures for Multi-Core" and in partnership with the Defence Science and Technology Group, through the Next Generation Technologies Program. Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award.

References

[1] T. E. Anderson. 1992. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*. 92–94. https://doi.org/10.1109/WWOS.1992.275682

[5] http://www.spdk.io/.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS '12*. ACM, 53–64.

[3] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP '09*. 29–44.

[5] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4 (Dec. 2016), 11:1–11:39.

[6] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. 2013. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, San Jose, CA. https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Blott

[7] Shekhar Borkar and Andrew A Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5 (2011), 67–77.

[8] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. 1996. An Implementation of the Hamlyn Sender-managed Interface Architecture. In *OSDI '96*. 245–259.

[9] Gregory D. Hill Albert H. Chen. 2015. *High Performance Network Multiplexing with IX++*. Technical Report. Stanford University.

[10] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst.* 32, 4, Article 10 (Jan. 2015), 10:1–10:47 pages.

[11] Tilera Corporation. 2014. TILE Processor and I/O Device Guide for the TILE-GX Family of Processors. (2014). http://www.mellanox.com/repository/solutions/tile-scm/docs/UG404-IO-Device-Guide.pdf

[12] Willem de Bruijn, Herbert Bos, and Henri Bal. 2011. Application-Tailored I/O with Streamline. *ACM Trans. Comput. Syst.* 29, 2, Article 6 (May 2011), 6:1–6:33 pages.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*. 251–266.

[15] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. 2012. TM2C: A Software Transactional Memory for Many-cores. In *EuroSys '12*. 351–364.

[16] S. Han, S. Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *OSDI'12*. 135–148.

[17] J. Howard. 2010. A 48-core IA-32 processor with on-die message-passing and DVFS in 45nm CMOS. In *2010 IEEE Asian Solid-State Circuits Conference*. 1–4. https://doi.org/10.1109/ASSCC.2010.5716540

[18] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. 2013. When Slower is Faster: On Heterogeneous Multicores for Reliable Systems. In *USENIX ATC'13*. 255–266.

[19] Tomas Hruby, Teodor Crivat, Herbert Bos, and Andrew S. Tanenbaum. 2014. On Sockets and System Calls: Minimizing Context Switches for the Socket API. In *2014 Conference on Timely Results in Operating Systems, TRIOS '14*.

[20] Tomas Hruby, Cristiano Giuffrida, Lionel Sambuc, Herbert Bos, and Andrew S. Tanenbaum. 2016. A NEaT Design for Reliable and Scalable Network Stacks. In *CoNEXT'16*. 359–373.

[21] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S. Tanenbaum. 2012. Keep Net Working - on a Dependable and Fast Networking Stack. In *DSN'12*. 1–12.

[22] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI'14*. 489–502.

[23] Antoine Kaufmann, iImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 67–81.

[24] J. Leverich. 2014. Mutilate: High-Performance Memcached Load Generator. https://github.com/leverich/mutilate. (2014).

[25] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *NSDI'14*. 429–444.

[26] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *ASPLOS'16*. ACM, 339–352.

[27] Ilias Marinos, Robert N. M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *SIGCOMM'14*. Article 9, 175–186 pages.

[28] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *NSDI'14*. USENIX Association, 459–473.

[29] Microsoft. 2008. Receive-Side Scaling. (2008). https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/introduction-to-receive-side-scaling.

[30] Bosko Milekic. 2004. Network Buffer Allocation in the FreeBSD Operating System. (2004). http://www.bsdcan.org/2004/papers/NetworkBufferAllocation.pdf

[31] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *SOSP '09*. 221–234.

[32] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *NSDI'13*. USENIX, 385–398.

[33] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 203–216.

[34] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4, Article 11 (Nov. 2015), 11:1–11:30 pages.

[35] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC'12*. USENIX, 101–112.

[36] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-application Library Operating Systems. In *OSDI'16*. 671–688.

[37] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. 2010. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *USENIX ATC'10*. 5–5.

[38] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. 1996. Netpipe: A Network Protocol Independent Performance Evaluator. *IASTED International Conference on Intelligent Information Management and Systems* 6 (1996).

[39] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *OSDI'10*. 1–8.

[40] Michael Stonebraker. 1986. The Case for Shared Nothing. *Database Engineering* 9 (1986), 4–9.

[41] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. 1993. Implementing Network Protocols at User Level. *SIGCOMM Comput. Commun. Rev.* 23, 4 (Oct. 1993), 64–73.

[42] Tilera. 2012. Tile processor architecture overview for the Tile-Gx series. (May 2012).

[43] G. Varghese and T. Lauck. 1987. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *SOSP '87*. 25–38.

[44] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *SOSP '95*. 40–53.

[45] David Wentzlaff and Anant Agarwal. 2009. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.* 43, 2 (April 2009), 76–85.

[46] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 43–56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

[47] Christopher Zimmer and Frank Mueller. 2015. NoCMsg: A Scalable Message-Passing Abstraction for Network-on-Chips. *ACM Trans. Archit. Code Optim.* 12, 1 (March 2015), 1:1–1:24.