Cosh: clear OS data sharing in an incoherent world

Andrew Baumann[†] Chris Hawblitzel[†] Kornilios Kourtis[§] Tim Harris[‡] Timothy Roscoe[§]
[†]*Microsoft Research* [§]*ETH Zurich* [‡]*Oracle Labs*[*]

Abstract

This paper tackles the problem of providing familiar OS abstractions for I/O (such as pipes, network sockets, and a shared file system) to applications on heterogeneous cores including accelerators, co-processors, and offload engines. We aim to isolate the implementation of these facilities from the details of a platform's memory architecture, which is likely to include a combination of cache-coherent shared memory, non-cache-coherent shared memory, and non-shared memory, all in the same system.

We propose coherence-oblivious sharing (Cosh), a new OS abstraction that provides inter-process sharing with clear semantics on such diverse hardware. We have implemented a prototype of Cosh for the Barrelfish multikernel. We describe how to build common OS functionality using Cosh, and evaluate its performance on a heterogeneous system consisting of commodity cache-coherent CPUs and prototype Intel many-core co-processors.

1 Introduction

This paper presents abstractions and mechanisms to allow processes on different cores to efficiently share data with clear semantics, regardless of whether the cores are cache-coherent or, indeed, share physical memory.

Computer architecture is in a period of change and diversification: core counts are increasing, and heterogeneous cores, such as GPUs, programmable NICs, and other accelerators, are becoming commonplace. Not only are memory hierarchies getting deeper and more complex, but the properties upon which much software relies, such as cache-coherent shared memory, do not always hold.

We argue that this presents an important challenge to system software, which frequently needs to share units of data such as file contents, images, bulk operands for computation, or network packets between tasks running on different processors. Moreover, the OS should facilitate such sharing between applications as well.

[*]Work completed while at Microsoft Research

The challenge is twofold. Firstly, to facilitate programming such platforms, the OS should provide a way to share data between cores whether or not they share memory, and whether or not any shared memory is cache-coherent. Secondly, to cope with both the complexity of memory arrangements in a modern machine, and the diversity of configurations across machines, the OS should provide this functionality in a manner which is *transparent*: clients should work efficiently without needing to be aware at compile time whether memory is shared.

In this paper we address this challenge with *coherence-oblivious sharing (Cosh)*, which abstracts the low-level distinctions between different forms of memory. Programs managing data with Cosh run efficiently on cores between which cache-coherent shared memory is available, or between which data can only be shared by using a DMA engine to copy it. Cosh aims to balance the requirements of *protection*, so that software can access only the data that it should, *performance*, so that using Cosh is never worse than using the hardware directly, and *portability*, so that the same software can run on a wide range of current and future hardware platforms.

Cosh provides an abstraction based on explicit *transfers* of bulk data. It is safe and natural for system programmers, yet nevertheless capable of unifying cache-coherent shared memory, non-cache-coherent shared memory, and cores which communicate only via DMA copies. For applications on a traditional, cache-coherent machine, Cosh exploits specially-mapped regions of shared memory, drawing on concepts developed for efficient interprocess data transfer in systems such as IO-Lite [21]. In contrast to traditional distributed shared virtual memory (DSVM) systems [e.g., 1], explicit transfer semantics make it clear at any point in the execution of a set of processes using Cosh which data they have access to, and rule out unsafe concurrent read/write sharing.

Our contributions are: (i) the design of the Cosh facility and semantics in Section 3, (ii) an implementation on real hardware covering a range of memory models in Section 4, (iii) showing how Cosh can be used to construct system services, including pipes and a shared file system

buffer cache in Section 5, and (iv) an experimental evaluation of Cosh's performance in Section 6.

2 Motivation

In this section, we argue that system-wide coherent shared memory is an unrealistic expectation in current computer systems, and this situation is unlikely to change. We conclude this from two trends in computer architecture.

Whither cache-coherence? The first trend is short-term: the use of GPUs and other accelerators in general-purpose computing is now commonplace. Discrete GPUs operate on private memory, which must be copied to and from the host RAM by DMA transfers. Recent integrated on-die GPUs have begun to relax this restriction, unifying system and GPU memory spaces, but the performance penalty for cache-coherent accesses to system memory is substantial [11, 12].

Current software architectures for GPUs target application developers through programming frameworks such as CUDA [20] and OpenCL [14], and interact with the OS only through low-level graphics-oriented driver APIs. Regardless of hardware support, the patterns for data movement and sharing between host cores and the GPU are limited to whatever high-level facilities the programming framework provides. As observed by Rossbach et al. [26], the lack of OS abstractions for managing GPU-based computation leads to significant performance penalties when more than one application uses the GPU, and much data is needlessly transferred. However, GPUs are moving away from the single-application model to become more amenable to management by system software. AMD's Fusion architecture is progressively adding support for general-purpose OS features such as virtual memory and context switching [25]. Finally, the utility of other accelerators such as programmable NICs and storage controllers has also been demonstrated [19, 29].

The second trend is longer term: it is possible that hardware-supported system-wide cache-coherent shared memory will no longer exist. As the number and variety of cores increases, the cost, complexity, and power needed to provided coherence at scale, and the achievable performance, suffer [13, 18]. Several prototypes eschew cache-coherence [9, 28] for this reason. A plausible outcome for future commodity systems is a progressive relaxation of hardware coherence. This may take various forms, such as "coherence islands", or "hybrid coherence".

Coherence islands are local tightly-coupled groups of cores (perhaps sharing a cache), with hardware cache-coherence available only within an island. Hybrid coherence models, such as Cohesion [13], provide a basic hardware coherence mechanism that may be dynamically enabled by software on a per-memory-region basis. Disabling coherence in such a model offers substantial gains in memory performance, and frees up coherence hardware (i.e., directory entries in a cache), also improving the performance of the remaining hardware-coherence regions.

Overall, Cosh targets three different scenarios we expect in the future, and from which we aim to abstract without sacrificing performance:

1. *Cache-coherent shared memory.* Cores access shared memory, and observe coherent data, according to the memory consistency model.
2. *Non-cache-coherent shared memory.* Cores access shared memory, but software must perform explicit *flush* (to evict modified data from a cache back to memory) and *invalidate* (to discard possibly-stale data from a cache) operations to ensure consistency.
3. *No shared memory.* Cores cannot access shared memory and rely on an indirect communication mechanism, such as a DMA controller, to copy data between their private memories. DMA is often not cache-coherent, so software may also need to flush and invalidate caches for DMA. Since DMA copies are typically high-bandwidth but also high-latency due to setup costs, it is advantageous to perform them in bulk. DMA is also asynchronous: a core initiates a transfer and continues to execute while it occurs.

An example hardware platform The platform we use to evaluate Cosh in this paper provides a good example of these trends. Architecturally, it sits somewhere between today's GPU-accelerated systems and the kinds of integrated, heterogeneous, non-cache-coherent platforms we expect in the future. It consists of a commodity PC coupled with a prototype Intel *Knights Ferry* co-processor.

The PC is based on a Intel DX58SO motherboard, and has one Intel four-core "Bloomfield" i7-965 CPU clocked at 3.20GHz. Each core has 256kB L2 cache, and the system has an 8MB shared L3 cache and 2GB DDR3 RAM.

Knights Ferry is the software development vehicle for Intel's Many Integrated Core (MIC) architecture [10]. It supports general-purpose parallel computing with 32 in-order cores running at up to 1.2GHz. Each core has four threads, a 512-bit SIMD vector unit, 32 kB L1 I- and D-caches, and 256 kB L2 cache. The co-processor is located on a PCIe card, with 2 GB GDDR5 memory on board.

The performance of Knights Ferry is not representative of the Intel MIC product; nevertheless, it has an important property for this work: memory on each co-processor

card is locally cache-coherent, and is therefore a coherence island. This memory is private to the co-processor, aside from regions which may be shared (non-coherently) with the host via PCIe windows. The card also includes DMA controllers for bulk copies to/from host memory.

In some respects, the Knights Ferry co-processor is similar to a GPU: the many multi-threaded cores are smaller and simpler than today's desktop and server processors, and are suited to data-parallel workloads that often benefit from GPU acceleration. However, unlike current GPUs, the Intel MIC architecture incorporates full-fledged x86 cores: it has all the features necessary to run a general-purpose operating system, and is thus a good basis for research into OS support for future platforms.

Implications for system software In response to the shifts in architecture, recent research has proposed adopting techniques from distributed systems, and structuring the OS as a message-passing "multikernel" system [2, 16, 30]. However, the availability of some shared memory (regardless of its coherence model) is an important distinguishing characteristic from classical shared-nothing distributed systems, and effectively exploiting it is key to achieving performance on multicore systems.

One alternative approach would be to provide an abstraction of coherent shared memory. Traditional DSVM systems [1, 23] implement such an abstraction over commodity networks, and there has been a resurgence of interest in shared virtual memory systems for non-coherent accelerators and co-processors [15, 17]. However, we show in this paper that the sharing semantics of the OS services we target are sufficiently restricted to admit a looser abstraction than DSVM, and OS functionality does not fundamentally require coherent shared memory.

By choosing an abstraction for transferring and manipulating bulk shared data that sits between the extremes of shared-nothing message passing and shared-always DSVM, we significantly simplify its implementation, and enable better performance.

3 Design of Cosh

In Cosh, we focus on the communication that arises in system services: structured sharing of bulk data between processes using OS APIs, rather than very fine-grained sharing of data structures between software threads.

Rather than providing a shared virtual memory model of threads, shared data, and locks, Cosh exposes explicit *transfers* of arbitrarily-sized data between processes running on different cores. Like message passing, transfers are always explicit; however unlike most message-passing systems, Cosh transfers do not imply copying. Implementations of Cosh can therefore provide this abstraction efficiently over different kinds of memory – with/without physical sharing, and with/without cache coherence.

Cosh can be used in multiple settings: transfers between kernel and user-mode, transfers between code running in different protection domains on different cores, or transfers between co-operating processes split over specialised cores and the main processor. In Section 3.1, we illustrate this with some of the settings we have in mind for using Cosh. We then introduce the design, principles, and API of Cosh, starting with a basic page-granularity transfer mechanism in Section 3.2, and then progressively extending it to permit efficient use of shared memory (Section 3.3) and transfer arbitrary byte-granularity aggregates (Section 3.4).

We make a number of design choices to enable efficient implementations: (*i*) Placement of received data is under the control of the Cosh implementation. This enables direct use of shared memory, without the need to copy data to a particular location at the receiver. (*ii*) When making a transfer, the sender promises how they will access the data (after transmission) and the receiver promises how they will access the data (after receipt). This identifies cases where data can be shared directly (e.g., if neither side will write to the data). (*iii*) The API distinguishes between co-operative use (where the sender/receiver are trusted to keep their promises) from protected use (where Cosh must ensure that broken promises will be detected). This distinction allows Cosh to avoid copying data or adjusting memory page protections when making transfers between co-operative processes with shared memory.

3.1 Use cases

We begin with a discussion of use-cases for bulk data movement and sharing that exist in common OS services. We consider not only traditional UNIX abstractions (such as POSIX I/O with its copy semantics) but also more flexible variations on the same interaction patterns. As has been observed by others [3, 4, 6], the performance gains permitted by the use of *move* or *share* semantics for I/O, rather than requiring copying, are substantial, and we do not wish to exclude these by design.

Pipes and related producer/consumer inter-process communication mechanisms transfer data that has originated in a single producer process to a single consumer process. Logically, data moves from producer to consumer. A pipe guarantees in-order delivery of byte-wise data, without preserving any other structure (i.e. writes may be coalesced). Extensions to classical pipes have

been proposed which relax the single producer and single consumer requirements [5].

Network sockets are often used as an inter-process communication mechanism, either with network protocols over a loopback interface or UNIX domain sockets. As with pipes, source data originates in a process writing to the socket. For connection-oriented sockets, the data may be queued, and is eventually moved to a single destination. For datagram-oriented sockets, a packet may be discarded or delivered to multiple recipients, depending on its destination. Guarantees on reliability, ordering, and packet boundaries vary between protocols.

File system I/O is an important form of structured inter-process communication and sharing. APIs for accessing files vary; we consider here only the classic read-/write interface, but note that in the absence of read/write sharing between processes, support for memory-mapped files is also possible in Cosh. Once written, data is long lived, and may be read multiple times by arbitrary processes until overwritten or deleted. Read and write operations are atomic and isolated: the results of a completed read cannot change, even if the file contents are subsequently written. Finally, strict POSIX semantics (which in practice are often weakened) require serialisability: a read operation which can be proven to occur after a write has completed must return the new data.

We observe that all of the above use cases, although commonly implemented using shared memory for the data, do not require it. Rather, all have explicit points at which data is transferred: APIs such as read, write, send and receive delineate logical transfers in data ownership, even if they are typically implemented by copying or remapping. This observation motivates the design of Cosh.

3.2 Terminology and core primitives

We next define the terminology we use in describing Cosh, and its core operations, along with the set of design invariants they preserve. In the sections that follow, we then extend our core design in two respects: decoupling rights from virtual memory permissions for cooperating domains (Section 3.3), and handling of arbitrary size (byte-granularity) data (Section 3.4). We focus on the viewpoint of a programmer using Cosh, rather than on the mechanisms used within a Cosh implementation. The API is intended to define functionality independently of any particular underlying implementation or hardware.

We call the endpoints between which communication occurs **domains**. For the purposes of Cosh, a domain has the following properties: (*i*) it defines a protection boundary; and (*ii*) memory may be shared arbitrarily within a domain. When Cosh is used in a traditional OS, a process meets these requirements.

Table 1: Transfer modes

Mode	Sender requires	Sender retains	Receiver gains
move	read/write	no access	read/write
share	read	read-only	read-only
copy	read	*unchanged*	read/write *copy*

A **buffer** is a contiguous region of memory, that is aligned to a multiple of the system's page size. Cosh is unaware of the structure of any data within a buffer. In typical target use cases, buffers are tens of kilobytes or larger. As we discuss in Section 4, this guides choices within our implementation.

Each buffer provides associated **rights** on the data within it: either no access, read-only, or read/write. A domain must promise that its memory accesses will respect the rights associated with a buffer. If the domain makes accesses that are not permitted, then the behaviour of Cosh is undefined for that domain. We describe later (Section 3.3) the exact guarantees that Cosh provides to other domains when a domain breaks this contract.

Cosh supports a **transfer** operation initiated by a sender domain. Each transfer specifies (*i*) a destination domain, (*ii*) a list of buffers, (*iii*) a **transfer mode**, and (*iv*) optional flags. A transfer consists of the following operations:

1. It checks that the sender holds the required rights on all the buffers to initiate the transfer.
2. It downgrades the sender's rights to the buffers, according to the transfer mode.
3. It provides the destination with a new list of buffers. These buffers contain the same data as the sender's buffers at the time of the transfer; its rights are determined by the transfer mode.

All sharing of data in Cosh occurs through the transfer mechanism. Depending on the transfer mode, and the functionality of the underlying hardware, a transfer may be implemented by mapping virtual memory pages, copying, or a combination of both.

The set of transfer modes is shown in Table 1. Each transfer mode specifies the minimum rights *required* by the sender for the transfer to succeed, the rights *retained* on buffers held by the sender, and the rights *gained* by the destination after a transfer. These are:

Move: A sender with read/write rights hands data off to a receiver, who acquires read/write rights. After the transfer, the sender loses all rights to the buffers. This mode is typical of producer/consumer arrangements such

as a processing pipeline where domains may modify data in-place as it passes.

Share: A sender with read-only or read/write rights to data passes it read-only to a receiver. If the sender had read/write rights before the transfer, then its rights are downgraded to read-only after the transfer. This transfer mode is useful when multiple domains share read-only access to data. We use this to implement a file system; it may also be used for multicast pipes, where a sender transfers the same data to multiple receivers.

Copy: A sender with read-only or read/write rights to data transfers a copy of that data to a receiver, who acquires read/write rights on the copy. The sender's rights are not affected. This is the only transfer mode that may result in both sender and receiver holding read-/write references to the data, however, whereas the other transfer modes may be implemented by page remapping, this mode guarantees that the resulting buffers behave as copies. In practice, the copy may be performed at transfer time, or lazily with copy-on-write techniques.

Design invariants These guided our design:

1. *A transfer preserves no structure beyond a vector of byte values.* To allow optimisations such as scatter-gather DMA, and handle possible variations in page size, we do not guarantee to preserve the structure of buffers in a transfer: for example, a transfer of two 8kB buffers may appear to the receiver as a single 16kB buffer, or (perhaps unlikely) four 4kB buffers.
2. *Rights may be downgraded by a transfer, but never upgraded.* Once a buffer has been shared, it remains read-only in all domains until deallocated. This prevents read/write sharing.
3. *It is always correct to implement a transfer as a copy of the data.* This allows DMA transfers when sender and destination do not share memory, and enables optimisations when it is more efficient to copy than to remap memory. A corollary is that *no domain should observe changed buffers without an explicit transfer.*

Cosh also provides a fundamental property: so long as sender and receiver perform the correct transfer operations, and access their data only as promised, then they cannot distinguish between execution with/without shared memory, and with/without cache coherence.

3.3 Weak vs strong transfers

Until now, we have described the transfer operation in terms of its effect on a domain's *rights* to access the buffers involved. We next define what guarantees Cosh gives when a faulty or malicious application attempts to access buffers to which it does not hold appropriate rights.

This is irrelevant for copies, but significant when the transfer is implemented by memory remapping: if we trusted all domains to behave correctly according to the API, it would be possible to map all memory read/write and implement transfer simply as meta-data updates and cache flush/invalidate operations. Conversely, a pessimistic assumption of malicious domains would require that virtual memory permissions be updated in sender and receiver on each transfer. As we will show later in Section 6, this has a large performance penalty for trusted domains (e.g., OS services such as a file system).

Cosh therefore allows the programmer to choose between **strong** and **weak** transfers. Strong transfers are the default case, with weak transfers available as an optimisation where trust allows. After a strong transfer, Cosh guarantees that no domain's virtual memory permissions exceed its rights to the transferred buffers; that is, no domain can observe a buffer to which it has no rights, or modify a buffer to which it has read-only rights. However, after a weak transfer, a domain's virtual memory permissions on shared buffers may continue to exceed its rights, until such time that a future strong operation downgrades them.

Weak transfers act as a hint to Cosh that permit it to defer changing virtual memory mappings if this will result in better performance; the implementation is free to ignore this hint. They are useful when sender and receiver have an appropriate trust relationship (for example, if the sender is a system service that is trusted by the receiver not to modify a buffer after its transfer, or if the receiver independently protects itself from such modifications). Note also that Cosh is not required to enforce changes to virtual memory permissions when it performs a strong transfer as a copy: strong transfers cannot be relied upon to catch bugs arising from incorrect memory accesses after a transfer, only to contain their effects.

Regardless of any prior weak transfers, rights are always strongly enforced upon (re)allocation of memory: Cosh guarantees that freshly-allocated buffers cannot be observed or modified by any other domain until transferred. Furthermore, when it is transferred, and a domain first acquires access to a buffer, Cosh guarantees that the domain's virtual memory permissions will not exceed its rights. Put another way, after a buffer is allocated and until it is reallocated, the set of domains that may observe its contents is limited to those that received it in a share or move transfer, and the set of domains that may modify them is limited to those that received it in a move transfer. This results in the following guarantees:

Table 2: Summary of aggregate API

`alloc(len, flags) -> agg`	Allocate storage for a new aggregate
`incref(agg)`	Increment reference count
`decref(agg)`	Decrement reference count and deallocate if zero
`getlen(agg) -> length`	Return length of aggregate
`getrights(agg) -> rights`	Return current rights on aggregate
`iter_start(agg, read|write, offset) -> iter`	Start iterator at the given byte offset within aggregate
`iter_next(iter) -> addr, length`	Return next portion of iterator
`iter_end(iter)`	Release iterator
`concat(agg1, agg2) -> agg`	Create aggregate which is the concatenation of the inputs
`select(agg, offset, length) -> agg`	Create aggregate which is a sub-region of the input
`find_related(agg, minrights) -> [agg]`	Find all related aggregates with at least `minrights`
`downgrade(agg, rights)`	Downgrade rights on aggregate
`transfer(agg, dest, transfer_mode, flags)`	Initiate transfer of aggregate to domain `dest`
`receive(src) -> agg, transfer_mode, flags`	Wait to receive incoming transfer

Strong move: all domains other than the destination are prevented from reading or writing to the buffers.

Weak move: domains who have previously received access in a transfer may retain read/write permissions on the buffers, but are trusted not to access them.

Strong share: all domains are prevented from writing.

Weak share: domains that previously received access in a transfer (possibly including the destination) may retain write permissions on the buffers, but are trusted not to write to them.

Copy: the destination receives a copy of the data, and may assume that no other domain can read or write it. Copy transfers are strong, since memory is not shared.

In the Cosh API, a weak transfer may be requested by the sender domain when initiating a transfer through an optional flag parameter. The destination is notified of the transfer type (including whether it is weak or strong) upon delivery, and should treat an inappropriate transfer mode as a protocol error by the sender.

3.4 Aggregates

Page-granularity buffers permit efficient implementations of transfer using virtual memory remapping. However, this constraint is ill-suited for some important target use-cases: many I/O interfaces operate at smaller granularities, such as bytes. Cosh therefore supports an aggregate abstraction, layered over the base primitives described above, for operations on arbitrary byte-granularity data.

An **aggregate** is an ordered sequence of buffer sub-regions, each of which has an arbitrary byte-granularity size and offset within its containing buffer. Each aggregate is local to a specific domain. An aggregate serves as a container for data that is being managed using the Cosh API. Like buffers, an aggregate has associated rights. An aggregate's rights change over time, but always apply to the entire aggregate (rather than to subsets, such as individual buffers). Cosh operations are defined so that the rights on an aggregate never exceed those of its component buffers.

A domain may transfer an aggregate to a destination, using the same transfer modes (Table 1). As with buffer transfer, Cosh preserves only the contents, not the structure of the aggregate: an aggregate consisting of many small buffer regions on the sender side may be delivered as an aggregate with a single contiguous buffer.

Aggregate API Table 2 summarises the high-level aggregate-based API. From the programmer's perspective, all operations manipulate the aggregate abstraction. Aggregates are reference-counted, and the data they contain is accessed using iterators.

Using the `concat` and `select` operations, it is possible to express modifications of data to which the domain has read-only access by constructing a new aggregate that combines portions of the original aggregate with portions of modified data. Our file system implementation (described in Section 5) exploits this.

Transfer uses a message-like API: the sender invokes a call to initiate a transfer, which may return before the transfer is complete. The destination independently invokes a blocking `receive` call, which returns the new aggregate and its metadata. As we later discuss, our use of the AC ("Asynchronous C") language [8] permits lightweight asynchrony for aggregate transfers without the need for extra software threads.

Related aggregates The `concat` and `select` operations create a new aggregate based on one or more existing aggregates in the same domain. We say that the new aggregate is **related** to the input aggregates; relationship is transitive and symmetric, and captures the fact that, within a domain, a set of aggregates may refer to the same buffers.

When an aggregate is transferred, Cosh downgrades the sender's rights on the transferred aggregate according to the transfer mode. However, since other related aggregates may also refer to the same buffers, we must also specify what happens to them as a result of a transfer. This choice has important implications for the usability of the API; we considered three alternatives:

1. The rights on all related aggregates are downgraded as a side-effect of the transfer, along with the transferred aggregate.
2. If there are any existing related aggregates whose rights exceed those to which the transfer would downgrade the transferred aggregate, the transfer operation fails.
3. The transfer operation downgrades rights only on the transferred aggregate, not related aggregates. If any related aggregates with excessive rights refer to buffers to be transferred, the affected portions of the aggregate must be transferred by copying (rather than remapping), to preserve the access rights of those related aggregates.

The first two options permit all aggregate transfers to be implemented by virtual memory remapping, but our experience shows that they are difficult to use in systems with non-trivial data flow or sharing between components, because the shared nature of related aggregates prevents local reasoning about the semantics of a transfer operation. In particular, it is undesirable for a local operation in one software module (a transfer) to affect an aggregate held by another module that may happen to refer to the same memory, or to be able to fail as a result of the actions of that module. The Cosh aggregate API therefore maintains local semantics for a transfer by requiring copies (or copy-on-write) where necessary. However, the API also allows the caller to request that the transfer fail in the presence of related aggregates, and includes a separate operation to locate such related aggregates. We expect that this would be used primarily for debugging.

4 Implementation

In this section, we summarise our implementation on the target system of x86 and Knights Ferry. To enable this prototype, we first ported Barrelfish [2] to run on our heterogeneous target system. This consisted primarily of: (*i*) porting the *CPU driver* (the per-core kernel-mode component of Barrelfish) to run on Knights Ferry cores, (*ii*) implementing a message-passing *interconnect driver* for sending short messages between host and co-processor, (*iii*) orchestrating the bootstrap of the system, and (*iv*) implementing a driver for performing DMA transfers between the host system RAM and co-processor memory. Our version of Barrelfish supports message passing between arbitrary processes on host and Knights Ferry cores, and shares most system services that do not rely on shared memory (such as naming) between host and co-processor. As we discuss in Section 7, Barrelfish provided a useful prototyping platform, but Cosh does not depend on it, nor its multikernel architecture.

Our prototype implementation consists of a **client library**, which implements the API and is linked into each program using Cosh, and a service daemon termed the **block manager** which manages the memory used by Cosh and mediates transfers. One block manager instance is responsible for each distinct pool of physical memory in a system, as described below. The client library and block managers communicate using the built-in message-passing facility of Barrelfish, which supports by-value transfers of small (several words) typed messages. They are implemented in AC [8], an extension of the C language which supports composable asynchronous I/O. The use of AC permits the code to be written in a simple nested style, using "blocking" remote procedure calls, and yet operate asynchronously without the cost, nondeterminism, and added complexity of multi-threading. Our initial implementation uses 1398 lines of AC code in the block manager and 1010 in the client library.

We chose to implement the block manager functionality as a separate service for simplicity and implementation expedience; it is not required by the design of Cosh. Indeed, this structure imposes a performance penalty for communication with the block manager, and is a potential scalability bottleneck. We see Cosh as core OS functionality which may ultimately be implemented in the kernel of a traditional OS, or, in a multikernel OS like Barrelfish, as part of the trusted monitor which runs on every core.

Block managers Block managers are the trusted service component of our Cosh prototype. One block manager instance serves each physical memory domain (regardless of cache coherence); for example, on our target system, we run one block manager for the host PC and one for each Knights Ferry card. Each core in the system has access to one "local" block manager, whose memory it can access via virtual mappings. Clients only communicate directly with their local block manager.

A **block** is a fixed-size region of memory of at least page granularity. The block size in our implementation defaults to 16kB. Where hardware support and application requirements permit, blocks may be shared by memory mappings in multiple domains. Each block is uniquely

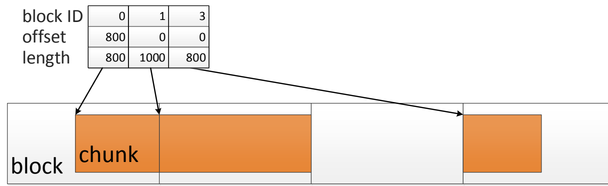


Figure 1: Internal structure of an aggregate

identified among the domains that might have access to it, and is managed by a single block manager.

Client library The client library is linked into each application using Cosh, and is responsible for implementing the API and communicating with its local block manager. Aggregates and the operations that manipulate them are implemented almost entirely within the client library. We describe the separation of concerns between the client library and block managers by introducing some further terminology:

A **chunk** is a portion of a block. It is identified by reference to the block and a (byte-granularity) size and offset within the block. A chunk may be as large as its containing block, but no larger. An aggregate is stored, transferred, and manipulated as a sequence of chunks. Within the client library, a linked-list representation is used, however, when communicating with the block manager, address-independent block identifiers, offsets and lengths are used.

Although it maintains each aggregate as a list of chunks, the client library transparently combines contiguous chunks in its implementation of the iterator API, and returns these larger buffers to the program. This permits programmers to assume, for example, that a freshly-allocated aggregate occupies a contiguous buffer, although it may be split over multiple blocks.

Figure 1 illustrates a sample aggregate that consists of three chunks (located in three different blocks). Since two of these chunks are contiguous, they will be presented to the application as a single buffer when it iterates over the aggregate. For each aggregate, the client library maintains the list of chunks making up the aggregate, the domain's current rights on the aggregate, the set of related aggregates, and a reference count.

The only API calls that communicate with the block manager are `alloc`, `decref` (when the reference count reaches zero), `transfer` and `receive`. All other operations are implemented locally in the client library, by manipulating aggregates. For example, `select` constructs a new aggregate by copying the meta-data for the appropriate chunks from the input aggregate, and marking the two aggregates as related.

Transfer When the application initiates a transfer, the client library first checks for appropriate permissions on the aggregate. It then transforms the aggregate to a form suitable for block-level transfer, by copying and coalescing chunks where required, updates its rights on the aggregate as required by the transfer mode, and contacts the local block manager to perform the transfer of blocks.

Some complexity arises from the fact that aggregates (and their chunks) are of arbitrary byte-granularity sizes, whereas the blocks that can efficiently be transferred by page remapping are much larger. As we described in Section 3.4, the behaviour of Cosh depends on a flag specified by the application. By default, portions of the aggregate which occupy the same blocks as any related aggregates are copied into new blocks before the transfer, and the rights on the related aggregates are unaffected. Alternatively, if any related aggregates have rights that exceed those they would be downgraded to by the transfer, the operation fails. The application may locate the related aggregates using the `find_related` API call.

At this point, the client library has constructed a vector of chunks to be transferred to the destination. It next contacts the local block manager, requesting it to perform the transfer and providing the vector. If the destination domain is a client of the same local block manager, the block manager adjusts permissions on the blocks upgrading them in the destination and downgrading them in the sender's domain as appropriate. If the client has requested a strong transfer, the block manager also downgrades the permissions on virtual memory pages mapping the block in any domains other than the destination. The block manager finally sends a message to the destination domain informing it of the transfer, and including the meta-data (block ID, offset and size for each chunk) from which the client library may construct the new aggregate.

As a performance optimisation, if the client requested a weak transfer, and the receiving domain already has suitable virtual memory mappings to all the blocks, there is no need to communicate with the block manager; instead, the transfer meta-data can be sent directly to the destination. We have not yet implemented this optimisation.

If the destination domain is not a client of the local block manager, it downgrades the sender's rights appropriately, determines the block manager responsible for the destination, and sends a message to the remote block manager to initiate a transfer. The remote block manager allocates local memory to receive the transfer, and then initiates a copy (e.g., on Knights Ferry, using the DMA controller). When the transfer is complete, the remote block manager notifies both the recipient and the source block manager, which releases an internal reference it was hold-

ing on the blocks for the duration of the transfer.

Our implementation assumes that data can be copied directly between any two block managers. Systems where this is not the case are out of scope for our initial implementation, but straightforward to handle, particularly if at most one intermediate copy (for example, through host system memory) is required.

5 Cosh Applications

This section describes the implementation of two common OS services using Cosh: pipes, and a shared file system cache. We include simplified code snippets to illustrate the use of the API, but elide error handling and other extraneous aspects.

Pipes We begin with a simple producer/consumer mechanism, equivalent to Unix pipes. A pipe moves data from a producer process to a consumer, however rather than copying data in and out of a kernel buffer, we use Cosh to transfer data directly from producer to consumer.

First, the producer allocates a new aggregate and fills it with data, or acquires it from some other source (such as the file system, below). It then initiates a *strong move* transfer of the aggregate to the consumer domain. It also releases its reference, since it no longer has access rights to the data, as shown below:

```
void pipe_write(wpipe *pipe, cosh_agg *agg) {
  cosh_agg_transfer(agg, pipe->dest, COSH_MOVE,
                    COSH_TRANSFER_STRONG);
  cosh_agg_decref(agg);
}
```

When the transfer completes, the consumer domain gains access to the aggregate. It is queued by Cosh for delivery to the next matching `receive` call, shown below:

```
cosh_agg *pipe_read(rpipe *pipe) {
  cosh_agg *agg; cosh_tx_mode mode; cosh_tx_flags flags;
  cosh_agg_receive(pipe->src, &agg, &mode, &flags);
  if (mode != COSH_MOVE || (flags & COSH_TRANSFER_WEAK)) {
    // protocol error by sender; tear down pipe
    cosh_agg_decref(agg); return NULL;
  }
  return agg;
}
```

The consumer may process the data, or pass it to other interface that accepts aggregates (for example, writing it to another pipe or file). When it is finished with the data, it decrements the aggregate's reference count.

The implementation of a producer/consumer mechanism such as a pipe using Cosh requires only a handful of lines of code in either domain, and offers the same semantics regardless of whether producer and consumer share cache-coherent memory, or lack shared memory entirely.

The astute reader may notice that our pipe lacks flow control: that is, in the presence of a slow or stalled consumer, a producer may continue to allocate memory and transfer data without bound. This can be avoided by using "out-of-band" inter-process messages between libraries in the producer and consumer to negotiate window sizes. In practice, we have not yet required it, since the Barrelfish messaging system provides its own flow control, but a portable Cosh application should not rely on this property.

File system and cache We next describe CoshFS, an in-memory file system and cache built using Cosh. Since our goal here is to exercise the Cosh API and gain experience with its use, rather than to build a high-performance scalable file system, our implementation errs on the side of simplicity, and maintains strict POSIX semantics: namely, (i) the results of a read operation are a stable snapshot of the file's contents at the time of the read, and (ii) a read always returns the results of a write (by any process) that occurred before the read. This has important implications for our implementation and our use of Cosh.

Property (i) requires that any memory used to store the results of a read operation remains immutable, even in the presence of subsequent writes to the file. We achieve this through the use of *share* transfers in Cosh, described below, which ensure that the file buffers are read-only.

Property (ii) requires serialising all read and write operations on the same file, and we achieve this by centralising all file system meta-data in a single service domain. As with distributed file systems, we expect that weakening this property would allow greater scalability.

Our implementation maintains a simple in-memory structure for file and directory meta-data. Each file is stored as an aggregate, to which the file system holds a reference. When a user domain writes to a file, it transfers the aggregate to the file system, and separately sends it a message containing the meta-data for the operation (i.e., file handle and offset).

Upon receiving the transfer for a write request, the file system checks that the transfer is *strong*, and rejects the request if not. The file system does not require write permissions to the data, so it does not matter whether the transfer is a *strong move*, *strong share*, or *copy*, since all three modes guarantee that other domains are prevented from modifying the buffer after the transfer completes. The file system then performs the write, by using the `select` and `concat` operations to construct a new aggregate for the modified file – it selects the portions of the original file before and after the write location, and concatenates them with the newly-written data:

```
void write_handler(client *cl, inode_t inode, size_t offset,
                   cosh_attach_ticket tkt) {
  // receive aggregate to be written
```

```
  cosh_agg *wdata; cosh_tx_mode mode; cosh_tx_flags flags;
  cosh_attach_recv(cl->domainid, tkt, &mode, &flags, &wdata);
  if ((flags & COSH_TRANSFER_WEAK)) {
    // protocol error by client; fail request
  }

  // validate inode, retrieve agg for file
  cosh_agg *file = lookup_inode(cl, inode);
  cosh_agg *newfile;
  if (...) {
    // handle special cases; e.g., write beyond EOF
  } else {
    // general case: construct aggregate for modified file
    size_t flen = cosh_agg_getlen(file);
    size_t wlen = cosh_agg_getlen(wdata);
    cosh_agg *pre = cosh_agg_select(file, 0, offset);
    cosh_agg *post = cosh_agg_select(file, offset + wlen,
                                    flen - (offset + wlen));
    cosh_agg *tmp = cosh_agg_concat(pre, wdata);
    newfile = cosh_agg_concat(tmp, post);
    cosh_agg_decref(pre);
    cosh_agg_decref(post);
    cosh_agg_decref(tmp);
  }

  // update stored aggregate
  update_inode(inode, newfile);
  cosh_agg_decref(file);
}
```

This code includes an example of a common pattern in Cosh: sending an aggregate concurrently with an IPC (or RPC) message with metadata for an operation. To support this pattern on Barrelfish, we wrote a small user-mode library on top of the aggregate API. It provides two primitives to transfer and receive aggregates: `attach_transfer` takes the same parameters as `transfer` and initiates the transfer of an aggregate, but also returns a *ticket*, which is an opaque integer (and thus may be sent in an IPC message) uniquely identifying the aggregate for that source and destination pair; its counterpart `attach_recv` takes a source domain ID and ticket, and blocks until aggregate with a matching ticket is received from the given domain.

When a user performs a read, the file system creates an aggregate for the appropriate portion of the file using the `select` API, and transfers it to the client as a *weak share* transfer. A weak transfer is appropriate here, because the clients trust the file system not to modify the file data, and because the file system is guaranteed that the client cannot receive a read/write mapping to a buffer that was only transferred as a *share*.

```
void read_handler(client *cl, inode_t inode, size_t offset,
                  size_t bytes, cosh_attach_ticket *ret_tkt) {
  // validate inode, retrieve agg for file
  cosh_agg *file = lookup_inode(cl, inode);

  // select portion to be read
  size_t len = min(cosh_agg_len(file) - offset, bytes);
  cosh_agg *agg = cosh_agg_select(file_agg, offset, len);

  // transfer to client
  cosh_attach_transfer(agg, cl->domainid, COSH_SHARE,
                       COSH_TRANSFER_WEAK, ret_tkt);
  cosh_agg_decref(agg);
}
```

The result of these transfer modes is that each domain reading a file receives a read-only reference to the same data. When those domains share memory, Cosh is free to avoid copies by mapping the same physical memory. Over time, the memory backing a file may become fragmented and sparse, as a result of reads and writes through many aggregates. This is handled by coalescing fragmented portions of files to new aggregates.

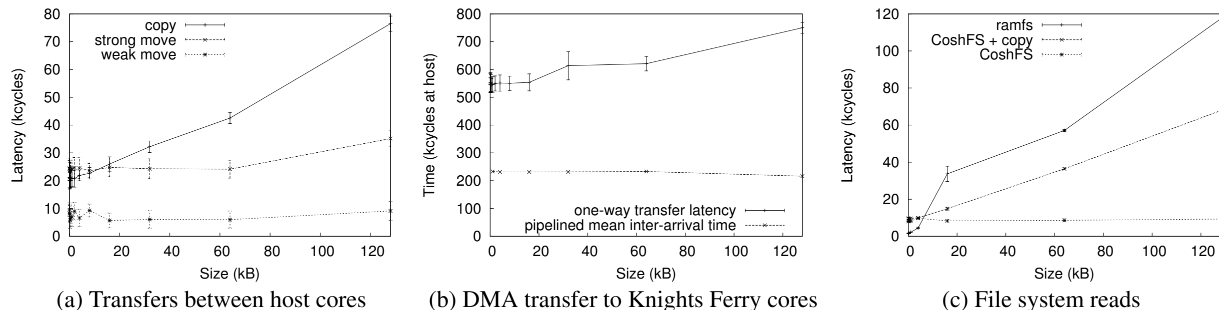
The structure we have described above is efficient when client domains and file system share memory, but can perform poorly when Cosh must perform DMA copies for each read or write. To mitigate this problem, CoshFS also includes a cache, akin to a shared (OS level) client-side cache in a network file system. Unlike a networked system, however, we assume both reliable message delivery and absence of failures, significantly simplifying the implementation. We cache both metadata and file data, thereby avoiding redundant communication with the central CoshFS service daemon, while maintaining strict POSIX semantics using exclusive write locks.

Support for legacy APIs The systems described above both use a "Cosh-aware" interface where user applications directly manipulate aggregates using the Cosh API. However, for legacy support, compatibility with traditional APIs using contiguous buffers may be desired. This can be trivially supported in a compatibility layer, at the cost of extra copies between aggregates and user buffers. We evaluate the cost of this copying in the context of the file system in the following section, however we note that in a fully shared-memory system, the number of copies imposed by the use of Cosh is no greater than those imposed by the legacy API itself, since an efficient Cosh implementation will copy only when it is faster than remapping virtual memory in this scenario.

6 Performance

Cosh primitives We begin with a microbenchmark showing the overhead of aggregate transfers. Two domains repeatedly transfer the same aggregate back and forth. Figure 2a plots the mean one-way latency (half the round trip time), measured in thousands of CPU cycles, for varying aggregate sizes and transfer modes. We run the two domains pinned to separate cores of the host PC when the system is otherwise idle, and repeat each transfer 100 times. Error bars show standard deviation.

The base cost for a transfer is approximately 6k cycles. This arises from our prototype implementation, requiring two RPCs rather than system calls for each transfer, which would be cheaper. The cost of a weak move is effec-



(a) Transfers between host cores (b) DMA transfer to Knights Ferry cores (c) File system reads

Figure 2: Results of microbenchmarks

tively independent of aggregate size, since the memory remains mapped read/write into both domains after the initial transfer. We also see that strong move transfers, which use page remapping, are generally cheaper than copy transfers, with the exception being small aggregates (here, less than 8 kB) where it is faster to copy.

These results are not surprising, but they confirm the value of supporting different transfer modes: for application uses such as file system reads that can tolerate the sharing semantics provided by weak transfers, they offer a significant performance improvement. Moreover, because it is always correct to implement a transfer as a copy, any case in which copying is faster than remapping can be exploited as a performance optimisation. This allows the implementation to use a variety of heuristics or online measurements to choose when to copy or remap memory.

We next measure the performance of transfers that cross the coherence and sharing boundary between host cores and the Knights Ferry co-processor. We use the same unmodified benchmark application as in Figure 2a, but vary the placement of one domain so that it runs on a co-processor. Since all transfers in this case are implemented as DMA copies, the transfer mode is irrelevant. Figure 2b shows the results, again measured as mean one-way latency in thousands of host CPU cycles.

DMA transfers have a significant fixed cost (approximately 550 thousand host CPU cycles, or 172 μs). However, much of this overhead is asynchronous and can be amortised through pipelining, as seen in the second dashed line on the graph, which shows the sustained mean inter-arrival rate for back-to-back transfers. This second line represents the achievable performance for a pipe or similar throughput-bound workload.

File system To evaluate our Cosh-based file system (CoshFS), we use as a baseline the default memory-backed filesystem in Barrelfish, named *ramfs*. We originally created CoshFS by modifying ramfs to use aggregates, so the two are directly comparable. The key difference is in the data transfer and API, since CoshFS supports file read and write operations using aggregates, whereas ramfs uses a more traditional API (i.e., `void *` and `size_t` arguments), and transfers data by copying it through a pre-established memory region shared with the client.

We first conducted a microbenchmark of repeated `read()` calls to the same file, shown in Figure 2c. This synthetic workload represents the ideal case for CoshFS, which uses Cosh to weakly share the buffer containing the file contents with the client, and thus automatically satisfies every subsequent read from the same previously-mapped buffer in the client's virtual address space. By comparison, ramfs, which always copies the data, performs better only for small reads where the overhead of communication with the Cosh block manager dominates. Beyond 4kB however, this cost is overtaken by the copying time, and CoshFS significantly outperforms ramfs. This can be seen as further motivation for the Cosh API property allowing all or part of any transfer to be implemented by copying: an optimised implementation of Cosh could exploit it to perform as well as ramfs for small reads, and do so without changing any code in CoshFS.

Figure 2c also shows a line labelled "CoshFS + copy", which simulates the cost of implementing a legacy (non-aggregate-based) `read()` API atop CoshFS, by copying the data once out of the aggregate into the application's buffer on the client side. For sufficiently large buffers, this still outperforms ramfs, which must make two copies for each read. This shows that although there are performance benefits to the use of the aggregate API, Cosh can also be used effectively in the implementation of legacy OS APIs.

For a more realistic test of CoshFS performance, since Barrelfish on MIC is not yet capable of supporting a full runtime environment, we replay the I/O generated by stitching and blending a sample panorama (using Hugin and Panorama Tools). This is illustrative of a multimedia processing workload that benefits from a co-processor

Table 3: Panorama stitching I/O trace

File system	Host (ms)	Co-processor (ms)
ramfs	145	*unsupported*
CoshFS	144	49742
CoshFS + cache		2464

such as Knights Ferry, and would exploit the shared virtual file system made possible by Cosh: rather than writing custom data movement code to transfer the input images to the co-processor, and then retrieve the results, the user can simply run the existing image processing application using the shared file system. The I/O trace was generated by `strace` on Linux; and then all file I/O calls were replayed, using the configurations shown in Table 3.

The middle column in Table 3 compares ramfs and CoshFS running on the host PC, where we find that both systems perform equivalently. This is due to a significant fraction of small I/Os (more than 80% of reads and writes in the trace are 4kB or less) where ramfs outperforms CoshFS, balanced by a "long tail" of large I/Os (up to 1MB in size) where CoshFS significantly outperforms ramfs. Given an optimised implementation of Cosh that copies rather than remapping pages for small transfers, we expect CoshFS to outperform ramfs by a larger margin. Nevertheless, the present result is encouraging: due to its use of Cosh, CoshFS is both more portable than ramfs and supports a much wider range of hardware.

We next move the replay program to one of the Knights Ferry co-processors, a configuration that cannot be supported by ramfs, since it assumes shared memory. We see that when accessing CoshFS on the host PC, the high cost of DMA transfers (shown in Figure 2b) for every I/O dominates, with the trace taking 50 seconds to replay. This is an unrealistic configuration, and motivates our addition of the cache (described in Section 5). By caching reads locally at the co-processor and avoiding the need to transfer intermediate data from the host, the overall replay time is reduced to 2.5 seconds. This is slower than when run entirely locally on the host for two reasons: first, the input images are initially un-cached and the DMA transfer time for these remains significant, and second, the trace replay runs serially on a single Knights Ferry core, which executes many times slower than a host core with its higher clock rate and out-of-order architecture.

Overall, our unoptimised Cosh prototype shows that the approach is tractable but does not yet achieve a desired level of performance. Nevertheless, Cosh enables significant added functionality while maintaining an API with clear and consistent semantics for applications, regardless of the hardware on which they run.

7 Discussion and future work

Alternative implementations We chose to implement our Cosh prototype on Barrelfish because the OS was a good fit for our target platform. However, nothing in Cosh requires Barrelfish, and it is instructive to consider the implementation of Cosh both for different OS environments and hardware. An implementation for a monolithic kernel such as Linux or Windows running on a cache-coherent system would place the functionality of the block manager in kernel mode, and make the Cosh API available for use by other kernel components such as the file system, and to user processes through system calls. It would support aggregate transfer to other kernels (e.g., running on a co-processor such as Knights Ferry) using copies, much as our block managers transfer aggregates between memory sharing domains. The resulting system would inevitably be less integrated than Barrelfish, but nevertheless, Cosh would facilitate sharing and communication between applications on the different kernels.

While we primarily considered general-purpose processors in the design of Cosh, the ability to perform all transfers as by-value copies makes it possible to implement a form of Cosh even for today's GPU architectures, where the GPU driver (and not the OS) is responsible for data transfers to and from the GPU. Given current GPU programming models, the initial use for Cosh on GPUs would likely be as a uniform transport for application-level data, that offers the same abstraction regardless of GPU vendor and core type. However, as GPUs acquire more general-purpose functionality (a trend we outlined in Section 2), it becomes possible to offer access to OS abstractions such as the file system from a GPU [27].

Future work Cosh is intended to support bulk data movement and sharing in OS APIs. A significant source of this data is I/O from device drivers, so a natural use of Cosh would be to integrate it further into the device stack. For example, reading (via DMA) from a storage device into an aggregate's buffers, moving the aggregate into the system's buffer cache, and sharing it with an application that reads from the file should be possible, all without requiring copies of the data (in the shared memory case).

One could also adapt a network stack to use Cosh as its data transport mechanism. Networking APIs partly motivated the design of byte-granularity aggregates, where headers may be appended to or removed from packets in-place. Even high-performance user-mode packet processing, such as netmap [24], fits within the Cosh API: it is possible to preallocate a large pool of aggregates, and use *weak move* transfers along a processing pipeline. Rather than deallocating an aggregate upon completion,

however, the established mappings could be maintained by using a further *weak move* back to the source domain (e.g. the NIC driver). This keeps all allocation and consequent memory remapping off the fast-path for a group of mutually-trusting domains. Thus, Cosh is flexible enough to match the performance of user-level shared memory mechanisms, yet still fall back to slower (but semantically equivalent) copying or remapping when required.

8 Related work

High-performance I/O As an OS abstraction for sharing and transferring bulk data, Cosh is related to high-performance I/O mechanisms for server applications [4, 6, 21, 22, 24]. This previous work has focused on optimising inter-process data transfer, particularly for producer/consumer scenarios. Significant performance gains are achieved through avoidance of copies, buffer management overhead and protection domain crossings. Cosh stands to benefit from many of the same optimisations. In contrast to Cosh, these previous systems operate within a fully cache-coherent shared memory environment. Cache coherence allows data and control information to be shared directly between the processes involved. The Cosh API permits sharing without cache coherence or shared physical memory, while nevertheless providing clear semantics.

The design of IO-Lite [21], in particular, inspired portions of the Cosh API. IO-Lite integrates the OS buffer cache with an efficient shared-memory bulk data transport mechanism. As with Cosh, IO-Lite features an API based on buffer aggregates, whose contents become immutable (read-only) upon a transfer out of the originating process. However, while IO-Lite used the mechanism for efficient snapshot semantics while using shared buffers for file reads, Cosh has a different purpose: to permit clean semantics over non-coherent or non-shared memory. Moreover, Cosh is more general than IO-Lite, in terms of its hardware support, target applications, and transfer modes. Finally, Cosh makes a distinction between weak and strong transfers. This enables an application to express the required level of protection, allowing the implementation to avoid unnecessary copying or page protection when enforced isolation is not required.

OS support for specialised cores Systems such as Hydra [29], Helios [19], PTask [26] and GPUfs [27] have demonstrated the benefits of OS support for accelerators.

Hydra and PTask target offload engines and GPUs respectively, with new application frameworks and runtime environments. Both transfer data across coherence and memory sharing boundaries, and the models adopted stem from the target hardware. PTask exposes a data-flow model, while Hydra uses dynamically-bound channels.

Like Cosh, GPUfs showed the value of giving GPU applications access to a shared file system. Because GPUfs is focused solely on file systems (rather than data transfer generally), the authors developed GPU-specific file system mechanisms such as parallel reads/writes and weakened consistency. A GPUfs-like file system could be implemented using Cosh as the data transfer component, which may address some of the performance limitations evident in our strictly-consistent CoshFS prototype.

Helios extends OS-level abstractions to “satellite kernels” running on heterogeneous cores. The semantics of inter-process communication in Helios derive from the Singularity OS [7]: all message transfers are expressed as a change in ownership from a source process to a destination, akin to a move in Cosh. Transfers are implemented using either shared memory or DMA copies (across coherence boundaries). Helios and Singularity rely on the use of type-safe managed code for applications and the OS, which allows move semantics to be cheaply enforced (ensuring that applications cannot observe or modify message contents after sending them) without the need to manipulate virtual memory permissions.

Cosh is more general than these systems, since it supports local sharing within a coherence boundary, and sharing between arbitrary (native) applications. In particular, share transfers permit processes to concurrently share a buffer, and weak and strong transfers allow applications to express the appropriate level of protection.

9 Conclusion

We presented Cosh, an OS abstraction for sharing and bulk data transport with clear semantics. Cosh does not depend upon shared memory, but can exploit it locally when available, regardless of whether it is cache-coherent.

We see an abstraction such as Cosh as a new portability layer, isolating OS services dependent upon data sharing from the details of hardware that provides it, which will become increasingly significant as hardware becomes more heterogeneous, and support for system-wide cache-coherent shared memory is diminished.

Acknowledgements

We thank Intel for access to the Knights Ferry hardware. We also thank the anonymous reviewers, Barry Bond, Galen Hunt, Ed Nightingale and our shepherd Robbert van Renesse for feedback.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), 1996.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *22nd ACM Symposium on Operating Systems Principles*, pages 29–44, Oct. 2009.

[3] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 277–291, 1996.

[4] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the OS traffic jam. In *INFOCOM*, 2008.

[5] W. de Bruijn, H. Bos, and H. Bal. Application-tailored I/O with Streamline. *ACM Transactions on Computer Systems*, 29:6:1–6:33, May 2011. ISSN 0734-2071.

[6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *14th ACM Symposium on Operating Systems Principles*, pages 189–202, 1993.

[7] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys Conference*, Apr. 2006.

[8] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy. AC: Composable asynchronous IO for native languages. In *26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 903–920, Oct. 2011.

[9] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International Solid-State Circuits Conference*, pages 108–109, Feb. 2010.

[10] Intel Corporation. Many integrated core (MIC) architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html, Apr. 2012.

[11] D. Kanter. AMD Fusion architecture and Llano. *Real World Technologies*, June 2011. http://www.realworldtech.com/page.cfm?ArticleID=RWT062711124854.

[12] D. Kanter. Intel's Sandy Bridge graphics architecture. *Real World Technologies*, Aug. 2011. http://www.realworldtech.com/page.cfm?ArticleID=RWT080811195102.

[13] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE Micro*, 31:42–55, 2011. ISSN 0272-1732.

[14] *The OpenCL Specification, Version 1.0*. Khronos Group, 2009.

[15] S. Lankes, P. Reble, O. Sinnen, and C. Clauss. Revisiting shared virtual memory systems for non-coherent memory-coupled cores. In *2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 45–54, 2012.

[16] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *1st USENIX Workshop on Hot Topics in Parallelism*, Mar. 2009.

[17] W. Liu, B. Lewis, X. Zhou, H. Chen, Y. Gao, S. Yan, S. Luo, and B. Saha. A balanced programming model for emerging heterogeneous multicore systems. In *2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[18] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *2008 ACM/IEEE Supercomputing Conference*, pages 1–11, 2008.

[19] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *22nd ACM Symposium on Operating Systems Principles*, pages 221–234, 2009.

[20] *CUDA Programming Guide*. NVIDIA, 2011.

[21] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18:37–66, Feb. 2000. ISSN 0734-2071.

[22] J. Pasquale, E. Anderson, and P. K. Muller. Container shipping: Operating system support for I/O intensive applications. *IEEE Computer*, Mar. 1994.

[23] J. Protić, M. Tomašević, and V. Milutinović. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, 1996.

[24] L. Rizzo. netmap: a novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference*, June 2012.

[25] P. Rogers. The programmer's guide to the APU galaxy. http://developer.amd.com/afds/pages/keynote.aspx, June 2011. AMD Fusion Developer Summit.

[26] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *23rd ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[27] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a file system with GPUs. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2013.

[28] C. Thacker. *Beehive: A many-core computer for FPGAs*

(v5). MSR Silicon Valley, Jan. 2010. http://projects.csail.mit.edu/beehive/BeehiveV5.pdf.

[29] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 179–188, 2008.

[30] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *ACM Symposium on Cloud Computing (SOCC)*, June 2010.