Optimus Prime: Accelerating Data Transformation in Servers

Arash Pourhabibi
arash.pourhabibi@epfl.ch
EcoCloud, EPFL

Siddharth Gupta
siddharth.gupta@epfl.ch
EcoCloud, EPFL

Hussein Kassir[*]
hussein.kassir@zhinst.com
Zurich Instruments

Mark Sutherland
mark.sutherland@epfl.ch
EcoCloud, EPFL

Zilu Tian
zilu.tian@epfl.ch
EcoCloud, EPFL

Mario Paulo Drumond
mario.drumond@epfl.ch
EcoCloud, EPFL

Babak Falsafi
babak.falsafi@epfl.ch
EcoCloud, EPFL

Christoph Koch
christoph.koch@epfl.ch
EcoCloud, EPFL

Abstract

Modern online services are shifting away from monolithic applications to loosely-coupled microservices because of their improved scalability, reliability, programmability and development velocity. Microservices communicating over the datacenter network require data transformation (DT) to convert messages back and forth between their internal formats. This work identifies DT as a bottleneck due to reductions in latency of the surrounding system components, namely application runtimes, protocol stacks, and network hardware. We therefore propose Optimus Prime (OP), a programmable DT accelerator that uses a novel abstraction, an in-memory schema, to represent DT operations. The schema is compatible with today's DT frameworks and enables any compliant accelerator to perform the transformations comprising a request in parallel. Our evaluation shows that OP's DT throughput matches the line rate of today's NICs and has ~60× higher throughput compared to software, at a tiny fraction of the CPU's silicon area and power. We also evaluate a set of microservices running on Thrift, and show up to 30% reduction in service latency.

CCS Concepts • **Computer systems organization** → **Architectures**; • **Information systems** → *Information integration*; • **Software and its engineering** → Cloud computing.

[*]This work was done while the author was at EPFL.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378501

Keywords Data Transformation, Hardware Accelerators, Microservices, Datacenters, Networked Systems

ACM Reference Format:
Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378501

1 Introduction

Deploying and maintaining online services in warehouse-scale computers [4] has become a task so complex that it has changed the best practices for software architecture [14, 22, 29, 31, 42]. Instead of single-binary monoliths, datacenter-scale applications are now best constructed as *microservices*, consisting of numerous self-contained modules communicating through Remote Procedure Calls (RPCs) or RESTful APIs [7, 9, 13, 14, 23, 45, 51]. Each microservice is written in the programming language best suited to its purpose, and uses the data format most natural to that language [13, 14, 26]. Therefore, inter-microservice RPCs must rely on the process of Data Transformation (DT) to convert data back and forth between the microservices' various formats.

The tremendous bandwidth of datacenter networks (e.g., 100Gbps in early deployment and 1Tbps on the roadmap) [48] and the body of systems research to optimize network hardware and software [1, 8, 21, 27, 33, 37, 40] have largely shifted the performance onus to server-side factors. To quantify, commodity NICs can already receive minimum-size Ethernet packets in just a few ns, and specialized network stacks already exist to keep up with such breakneck speeds [6, 27]. Additionally, as end-to-end transfer latencies approach the electrical limitations of propagation and switching [15], improving the performance of network communication has led to a "hunt for the killer microseconds" across the entire datacenter system stack [5]. We claim that because networks and protocols have already taken great strides forwards, DT is logically the next step to optimize and prevent from inhibiting the performance of network communication.

In theory, enforcing a single data format would eliminate the need for DT and enable all applications to communicate at the NIC's line rate. But in practice, microservices require an intermediary DT layer to ensure interoperability between each communicating pair of microservices. Currently, DT is performed exclusively by software, which expresses each transformation in long sequences of instructions in the CPU's ISA. Due to the inherent limitations of ISAs to express transformations and the CPU's ability to execute them, DT is already the bottleneck of inter-microservice RPCs.

We have observed that improving DT software's performance is difficult because each transformation executes copious dynamic instructions whose performance is hindered by data-dependent control flow. Even optimistically assuming five emitted instructions per output byte and perfect control speculation, transforming a $300B$ message (representative of common datacenter RPCs [33, 41]) requires upwards of $700ns$. This difficulty is exacerbated for more complex transformations, which require many more instructions per byte. Furthermore, even though each field of a message can be operated on in parallel, the transformations are fine-grained enough that synchronization costs limit any benefits from parallelization using threads. Even in an optimistic scenario, DT latency is already comparable to state-of-the-art protocol processing latency [27] and merits attention to reduce it.

In this work, we focus on alleviating the DT problem by first designing an abstraction for representing parallel transformation tasks, and then architecting a data transformation unit to unpack the tasks and perform them. Our key insight is that the parallelism amongst a message's fields is *implicit* when expressed in a CPU's ISA, but is much better represented by an *explicitly* parallel abstraction. To express this explicit parallelism to our accelerator, we make use of an in-memory schema that is created by the framework as the message is being built. This important design decision also means that our accelerator is applicable to any DT framework that generates this schema. We emphasize the importance of our architecture being general-purpose as it helps justify the investment in specialized hardware, which must apply to a variety of applications to be worth deploying. Nowhere is this challenge more pronounced than in the datacenter, where workloads have been shown to demonstrate significant diversity [4, 14, 28]. However, because our architecture is intended to be general-purpose, it boosts the performance of any datacenter-deployed microservice that depends on DT; we therefore claim that such an accelerator is an ideal candidate for inclusion in future server chips.

Finally, we implement and synthesize a concrete DT accelerator, and evaluate it against a set of representative DT tasks, frameworks, and microservices. Our accelerator outperforms commodity CPU cores by up to ~60× when transforming $300B$ Protobuf messages, with 2075× greater performance per watt. When applied to microservices running on Thrift's RPC stack, our design shows up to 30% reduction in overall service latency.

To summarize, we make the following contributions:

- An improved abstraction for transformation tasks: an in-memory schema which is parsed and parallelized by dedicated hardware.
- The design and architecture of an integrated data transformer, *Optimus Prime*. Our design goals are based on the constraints of modern server systems and identify the key characteristics for any DT accelerator.
- A concrete implementation of *Optimus Prime* which is compatible with today's DT frameworks and performs at the line-rate of today's 40Gbps server NICs, and continues to scale up to the bandwidth capacity of the server's on-chip network.

The rest of the paper is organized as follows: We first motivate the need for DT acceleration (§2). We then describe the design space for a data transformation accelerator (§3), followed by a description of our concrete implementation, named Optimus Prime (§4). Next, we detail our evaluation methodology (§5) and results (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Why Accelerate DT?

Microservices have emerged as a promising paradigm to build modern online services at datacenter scale, as they provide significant benefits for scalability, fault tolerance, reliability, and development velocity [4, 14, 26]. A microservices architecture consists of fine-grained software components with enforced modularity, interconnected through APIs such as RPCs or REST [14, 26, 44]. Because code structured in this fashion is often written in different languages with their own data formats, RPCs that cross format boundaries must perform *data transformation* (DT) to and from the desired format. Developers commonly use frameworks such as Google's Protocol Buffers (Protobuf) [19, 28] or Facebook/Apache's Thrift [3] for DT. These frameworks provide a structured way for applications to define the format of each message, and generate the code required to transform that message to and from the wire representation.

To illustrate the essential nature of DT, we present an example software stack in Fig. 1, where App 1 performs an RPC to App 2 with a `Person` object as its argument. App 1 first invokes the code generated by the DT framework to serialize the object into its wire representation. The binarized buffer is then passed down the stack to the RPC layer. When the RPC arrives at App 2, the same procedure is performed in reverse, where the object is deserialized into App 2's format. Critically, this step takes place for every network message

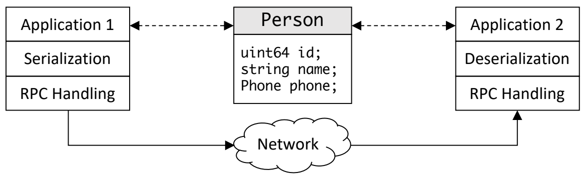


Figure 1. Two applications communicating using RPC.

between microservices, even those using the same data format, as the data must be flattened into a byte-stream at the sender and unflattened on the receiver side.

2.1 The Need for Faster DT

DT latency was long hidden behind slow network interfaces and protocol processing overheads. However, focused evolution efforts in the networking and systems domains have removed much of the overhead from the critical path of RPCs. Therefore, we claim that DT in its current form will inevitably dominate the future latency of inter-microservice RPCs. Furthermore, as the processing times of the microservices shrink to the μs-scale [5], DT will soon account for a considerable fraction of end-to-end latency. In fact, a recent study has shown that RPC and Protobuf software accounts for ~12% of total CPU cycles in Google datacenters [28]. We studied three microservices from DeathStarBench [14], and observed up to 30% of service time being spent in DT.

To quantify this claim, Fig. 2 illustrates the impact of improved network fabrics and protocol stacks on the latency of an RPC with a 300B payload, by comparing protocol processing time to DT time. The wire time is relatively small ($< 200ns$) and as such is not considered. The DT time is based on (de)serialization of Protobuf messages we used in our evaluation (§5), and protocol processing latencies for TCP and eRPC are taken from IX [6] and eRPC [27], respectively. In 10$Gbps$ commodity network stacks running TCP (e.g., in AWS [2]), protocol processing forms the dominant component of RPC latency. Combining faster fabrics [15] with optimized user-level software stacks [6, 27, 40] has resulted in sub-1μs protocol processing latencies [27, 33]. Therefore, with 40$Gbps$ fabrics and user-level RPC protocols, DT dominates the communication latency.

Finally, as silicon density scaling has slowed down in recent years (to as low as 17% annually), architects can no longer rely on the speedups historically granted by CMOS scaling to accelerate DT and match the rate of future NICs. To demonstrate, we used an Intel Xeon X5670, and measured the (de)serialization throughput of the messages in our evaluation (§5) as $100 - 300MB/s$. This is already an order of magnitude less than commodity 10$Gbps$ NICs and will fall further behind future higher-bandwidth NICs. Our goal is to eliminate DT as an inter-microservice communication bottleneck and boost the overall rate at which microservices can perform RPCs. We now present a walkthrough of the software DT, explain its limitations, and present the key insight enabling us to achieve DT at rates matching the NIC.

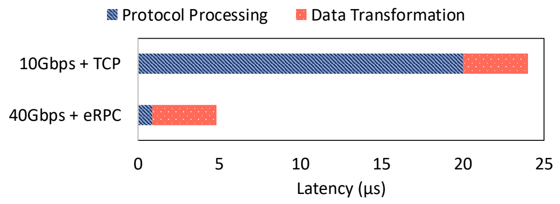


Figure 2. Breakdown of communication latency.

2.2 Data Transformation Walkthrough

As DT operations are similar across frameworks, we use Protobuf as a reference framework throughout this paper. For brevity's sake we focus on serialization operations, because the process of deserialization is similar but performed in reverse. The serialization process converts objects to a series of keys and values. Listing 1 shows pseudo-code for the serialization function. We use Fig. 3 to aid our explanation — it shows the fields of a basic `Person` object, and its final binary wire representation.

To serialize a `Person`, each field is individually transformed using the `serializeField` function based on its type. The output of each field contains a key (aka. tag), acting as an identifier for the field, and the serialized bytes of data. For example, the second field in Fig. 3, `name`, has its tag set to `0x12`, which comes from its type, 2 representing a `string`, and its field ID which is 2. Because the third field is an embedded message called `Phone`, `serializeField` recursively calls `serialize`, and all the output bytes corresponding to this message will be placed into the output stream following the tag. The `Phone` message is shown in Fig. 3 as an ellipsis.

For some fields, such as `float`, the source data is directly copied, but for others the source is completely transformed before being written. These complete transformations are the most challenging and compute intensive operations. For the rest of this section, we use variable length integer encoding (known as `varint`) as an example, but emphasize that all frameworks contain these types of transformations. The wire representation for the `id` field is called a `varint` in Protobuf, an encoding which depends on the data value. Only the number of bits required to encode the value (e.g., 32 bits to represent the value `123456789`, even though the language specifies 64 bits) will be sent on the wire. In order to signal to the receiver that there are more bytes to be processed which represent this integer value, the upper bit of each is reserved for the *continuation bit*, and is set to 1 if there are more bytes to come. These bits are shown as red and underlined in Fig. 3. Reading from left to right, each continuation bit would signal the receiver to "keep reading" as there are more bytes to come. The receiver stops processing the `varint` when it reaches a continuation bit with the value of 0.

2.3 Software DT's Bottlenecks

The process of transforming data on CPUs has two critical limitations, both inherently connected to the use of the ISA as an abstraction to represent the underlying operations.

Listing 1. Serialization Pseudo-code.

```
serialize (byteStream target):
  for (field f from 1 to N):
    serializeField(f, target)

writeVarInt64 (uint64 value, byte* target):
  while(value >= 0x80):
    *target = value | 0x80
    value = value >> 7
    ++target

  *target = value | 0x80
```

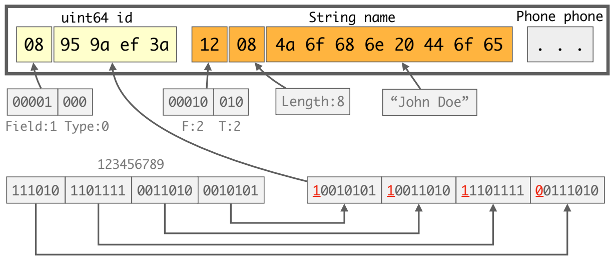


Figure 3. Sample Person object in Protobuf binary format.

Performing DT on CPUs entails a high instruction count per serialized field and relies on implicit instruction-level rather than explicit field-level parallelism. Additionally, DT is so fine-grained that it is unable to benefit from parallelization with software threads due to synchronization costs.

The cause for instruction bloat is the fact that many format encodings (e.g., `varint`) require performing an operation on each byte of the source data. In Listing 1, we show pseudo-code for transforming a 64-bit integer to a `varint`. For each byte of the input value, the code performs a branch to check whether or not the value is large enough to require the use of this byte. Then, bitwise operations are performed to isolate the correct byte and write it to the output buffer. On a commodity ARM X-Gene server with the microbenchmark used in our evaluation (see §5), we measured 25 dynamic instructions per byte, adding up to thousands of instructions per message. Executing this many dynamic instructions caps the achievable transformation throughput at roughly 1Gbps.

Improving the performance of serial instruction streams requires boosting the CPU's IPC. Unfortunately, transforming fields such as `varints` results in a data-dependent branch per byte; these branches are known to be difficult to predict and limit the achievable IPC by causing pipeline flushes. The limited success of control speculation when applied to data-dependent branches has also been observed by prior work studying ETL workloads for data cleaning and ingestion [12]. Although classical microarchitecture techniques such as predication [39] would help the performance of `varint` encoding, today's CPUs only support partial predication and lack the ability to perform conditional stores [30].

Finally, we do not expect parallelizing fields with software threads will yield significant improvements due to synchronization costs. Using a simple lock to signal work completion costs a minimum of 200*ns* per thread [10], which becomes equal to the time of performing the work serially with only five threads. In principle, each field in `Person` could be independently transformed if the hardware is made aware of each field's type and memory location. In that case, while the `varint` encoding is being performed, the `name` can be copied and the `Phone`'s data can be fetched. Unfortunately, neither software threads nor CPU ISAs are the right form to represent this parallelism between fields.

Designing an effective abstraction that explicitly represents this parallelism is key to accelerating DT. Serial instructions are the wrong abstraction to expose these types of independent operations, because the problem is inherently parallel. Therefore, because CPU-centric DT continues to be bound by the limitations imposed by the ISA, we argue that accelerating DT requires both hardware and software to be co-designed around a new parallel abstraction that replaces the ISA. The next section presents our hardware/software co-design for rapid and flexible DT.

3 Design for DT Acceleration

In this section, we describe the design space for a data transformation accelerator (DTA) that we argue must be comprised of an explicitly parallel DT abstraction, and the requisite hardware to perform the underlying operations. We lay out the design of a DTA prioritizing the following three goals. First, a DTA should perform DT at NIC line rate. Second, a DTA should be compatible with existing DT frameworks and programmable to allow compatibility with future data formats. Third, a DTA should have minimal impact on existing server architecture, limiting deployment cost. To achieve these goals, we seek to answer the following questions: i) what interfaces should the accelerator have with the software framework and the server system, ii) what components should be used as building blocks, and iii) where should the accelerator reside in the server?

3.1 New Abstraction: Transformation Schema

Based on our analysis in §2.3, we claim that accelerating DT requires an abstraction that expresses the parallelism inherent in transforming independent fields and solves the bottlenecks of expressing transformations in traditional ISAs. When a transformation is compiled into a CPU's ISA, field-level parallelism is only unlocked if the core is able to speculate far enough ahead to issue instructions that actually operate on two fields simultaneously. Our experiments in §2.3 show that doing so requires many hundreds or thousands of instructions, greatly exceeding the practical limits on a CPU's speculative state. An efficient DT abstraction

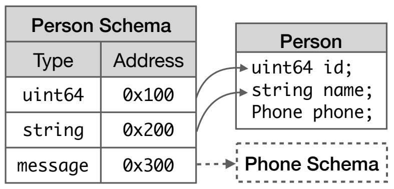


Figure 4. Sample Person object and its schema.

therefore requires representing transformations in an explicitly parallel fashion. The hardware can then unpack the field-level parallelism and enjoy the performance benefits.

The critical observation that leads to our novel transformation abstraction is the transformation on each field is completely described by its type (thus identifying the operation the hardware must perform) and the address of the input data. Therefore, a data structure containing these two pieces of information for each field is the leanest abstraction required to express all of a message's transformations. We call our DT abstraction the *schema*, which resides in memory and holds the type and address of each field. The schemata are generated by the application, and passed to the accelerator to invoke a new transformation. The software framework (e.g., Protobuf) needs to be modified to create the schemata during the process of creating the message. This can be done by updating the *setter* methods generated by the Protobuf compiler, to populate the schema's `address` field as well as the message's value. Fig. 4 shows an example of the schema for a `Person` after each field has been initialized.

Our schema design achieves both goals: first, it enables the hardware to operate on each field in parallel by scanning the schema, accessing the data to be transformed, and performing the requested operations. Second, it enables any framework to use the accelerator; the only requirement is that it updates the schema while creating the message. We now present the interface design between a DTA and the software stack.

3.2 DTA Interfaces

In the left half of Fig. 5, we see a traditional multi-core server system, with a number of cores connected by an on-chip network (NoC). The DTA's interfaces are constrained primarily by the μs-scale latency requirements of transformation tasks (§2), and thus we architect its invocation and data access paths to minimize latency. Kernel drivers and interrupts are too slow in this climate and therefore we use interfaces that allow user-level polling. Additionally, as the DTA needs access to the schemata and messages, it requires access to the server's virtual memory system.

Given such access to the virtual memory system, the DTA can use regular pointers for schemata and messages, avoiding wasteful copies. The DTA exposes a set of internal registers to the system software, which we map to I/O virtual addresses (IOVAs) in each process' address space to enable kernel-bypass and minimize latency [6]. Applications use

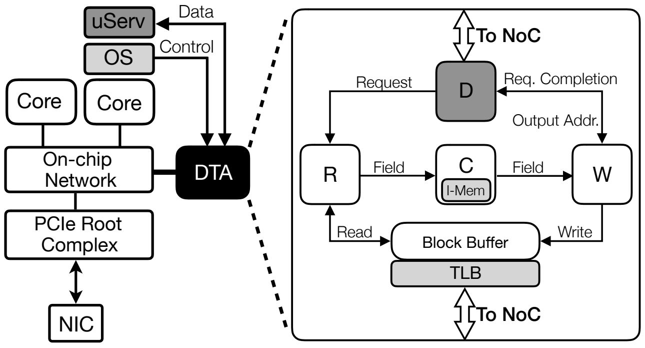


Figure 5. Architectural overview of a DTA. Light grey structures are configured by the control path, and dark grey structures directly communicate with the application.

memory-mapped I/O (MMIO) writes to these IOVAs to request new transformations, and repeated MMIO reads to poll for completions. Each request contains pointers to the schema and the output buffer for this transformation.

For an application to begin using the DTA, it performs a system call that returns a private context containing: (i) a set of per-core memory arenas where all messages from the application must be constructed, and (ii) the I/O virtual addresses where new requests are to be submitted. It is common for DT frameworks to use arena-based memory management [17], which follow the principles of user-level allocators like `jemalloc`. Because software already builds its messages in these arenas, the system call provides the arenas' virtual addresses to the OP so it can access the messages to be transformed. While there are rare cases where the application has pre-populated responses in its internal data structures, we focus on the general case where the application creates a new response message for every request.

To provide compatibility with a variety of DT frameworks, our DTA also has an interface for applications to program custom transformations. Upon requesting a new transformation context, the application also has the option to issue system calls to program custom operations into the DTA itself. We now present the internal building blocks of our DTA and how they implement the above interfaces.

3.3 Building Blocks

The right half of Fig. 5 shows the specialized hardware components comprising the DTA. We start with the Dispatcher (labelled D) component as it is responsible for interacting with the server's cores. The Dispatcher contains the DTA's internal registers which are read/written by the cores when invoking new transformations. Upon receiving a new request, the Dispatcher unpacks the schema and output buffer pointers and sends them to the transformation pipeline.

Conceptually, the DTA is architected as a decoupled access-execute pipeline [43], to deal with the challenge of specialized hardware components that can perform transformations an order of magnitude faster than data can be accessed from the memory hierarchy. Therefore, our DTA contains a pipeline of three components dedicated to parsing the schema and accessing data from the server's memory, performing transformations, and writing back results.

We first describe the specialized Converters (denoted C in Fig. 5), and how they achieve complex transformation operations (e.g., `varint` encoding) in a few cycles.

Recall from §2.3 that serializing a `varint` requires at least one branch, two arithmetic operations, and one memory access per byte. By extracting each byte from the source data independently, performing the range checks, and inserting the correct continuation bits, specialized hardware can achieve this complex operation in a single cycle. Each such operation is read from a small instruction memory (I-Mem) which resides in the Converter. By designing the DTA's Converter around such specialized operations, the accelerator can attain transformation throughput at higher rates than traditional cores.

In keeping with our goal that the DTA's architecture should apply to various DT frameworks, we also make the Converter's I-Mem programmable by system software. This means that the DTA is still usable for rare transformation operations, and is forward-compatible with new software. Such custom operations will inevitably have reduced transformation performance due to the return of ISA limitations, but they will still reap the field-level parallelism enabled by using our schema. If software wishes to customize the I-Mem's contents for new transformations, it supplies an argument to the system call which creates each application's transformation context (see §3.2).

To overcome the problem of idle Converters in the presence of memory accesses to the schemata and messages, we create two decoupled components responsible for accessing the memory hierarchy. These two components are the Reader and Writer (denoted R and W in Fig. 5 respectively), which access data and stream it to/from the programmable Converter. Our schema also enables the Reader to perform multiple parallel memory accesses without requiring speculation, as each field's address is explicitly written in the schema by software.

All memory accesses are performed by means of a non-coherent Block Buffer which acts as a scratchpad for the Reader and Writer, translates the virtual addresses of each field, and issues the corresponding reads and writes to the server's NoC. Once the transformation is complete, the Dispatcher's registers are updated and the core will see the completion with its next MMIO read. We next discuss the physical placement of DTA in the server.

3.4 Physical Location

Placing the DTA off-chip near PCIe-attached network interface cards (NICs) offers the lowest cost and least intrusive design point. However, the ~1μs latency of the PCIe interconnect [36] quickly becomes an obstacle for common nested messages (e.g., `Person`), due to the multiple PCIe roundtrips to fetch the nested message's pointers. Therefore, we focus on the tradeoffs inherent to an on-chip DTA, and discuss the following options: a private DTA co-located with each core, or a shared DTA that is placed on one of the chip's tiles.

These two choices expose a critical tradeoff between transformation latency variability and silicon provisioning. In the case where a DTA is co-located with each CPU core, it shares the core's L1 cache and TLB, eliminating the need for the Block Buffer component. However, private DTAs will require the Reader, Writer, Converter and Dispatcher to be replicated. Effectively, choosing private DTAs costs more silicon but eliminates variability in transformation latency, which is attributable to the NUCA architecture of the server's LLC. In contrast, attaching the DTA to the NoC as a shared component accepts the variability but attains more efficient silicon provisioning. We claim that despite the higher memory access latency and variability, the DTA should be shared due to the fact that the silicon costs of private DTAs quickly add up with increasing core counts. Our DTA is therefore shared and sits at the chip's edge, as shown in Fig. 5.

Summary: To achieve all the design goals (i.e., transformation at network line-rate, limited programmability, and integration simplicity) for an effective DTA, our design contains the following three essential characteristics: 1) The use of a powerful schema which uses simple type identifiers and memory addresses, enabling field-level parallelism and the ability to overlap data access latency. 2) Specialized hardware converters which can perform data transformations in a handful of cycles, and support a variety of operations defined by the software. 3) On-chip integration for low-latency access to the server's virtual memory system.

4 Optimus Prime

In this section we present Optimus Prime (OP), our implementation of a DTA which follows the principles in §3. Figure 6 displays the microarchitecture of OP, comprising its five major components: Dispatcher (§4.1), Block Buffer (§4.2), Reader (§4.3), Converter (§4.4), and Writer (§4.5). As we describe each component, we walk through the process of *serializing* a message. A similar process applies for *deserialization.*

4.1 Dispatcher

The Dispatcher receives transformation requests from the cores and notifies the corresponding core upon completion. It contains a set of dedicated control registers which CPU cores access through MMIO to request new transformations (§3.2). Each request includes a pointer to the schema to be transformed, a pointer to the output buffer where data has to be written, a pointer to the serialized buffer (for deserialization only), and a valid bit. When a new request arrives in the control registers, the Dispatcher controller passes the schema

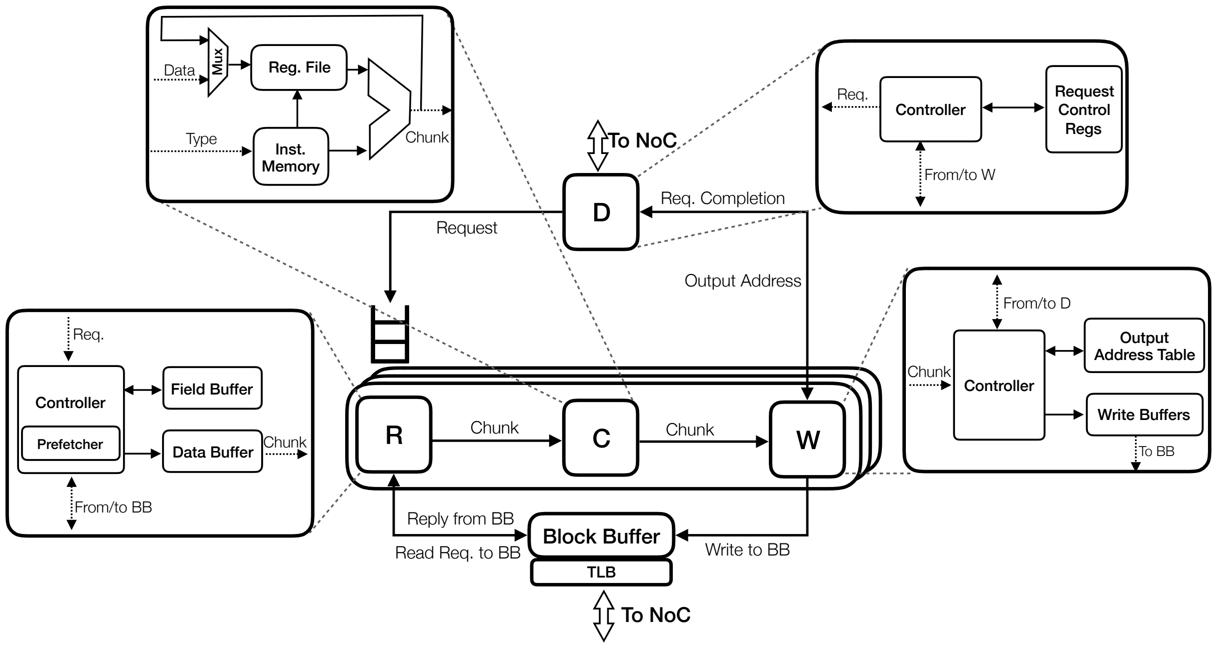


Figure 6. Overview of the microarchitecture of Optimus Prime.

pointer to the Reader and output buffer pointer to the Writer. Upon request completion, the valid bit is cleared and the CPU core will determine the transformation is completed with its next MMIO read. In our implementation, transformations are synchronous in nature; therefore, a core waits for a request completion before it issues another. Asynchronous transformations can also be implemented by writing each request to a different control register and polling each one.

4.2 Block Buffer

OP has a virtually-indexed, virtually-tagged Block Buffer, which is not coherent with the rest of the on-chip hierarchy. Synonyms are resolved by tagging each entry with the core ID associated with the transformation. If a data request to the Block Buffer results in a miss, the Block Buffer issues an explicit read request using the cache coherence protocol. The Block Buffer has a TLB, which contains the virtual-to-physical translations for per-core memory arenas where applications construct their messages. The OS allocates and pins a per-core arena at initialization time for each application and fills the TLB with the translations (§3.2).

Due to their latency-critical nature, microservices are likely to run on dedicated cores. As such, the TLB has as many entries as cores, is directly indexed by core ID and maintains the translation for as long as the microservice is active. Such a direct-mapped table has a small silicon footprint even with hundreds of cores. As the cores create every message in their private arenas whose translations are preinstalled, the TLB never misses. The total amount of pinned memory for the arenas is also relatively small given that modern servers integrate hundreds of GB of DRAM [2]. Filling translations into the TLB at initialization time is a low-cost operation required only in the case of a context switch.

4.3 Reader

The Reader parses the schema that comes from the Dispatcher, fetches all the fields from memory (via the Block Buffer), and sends them to the Converter. The Reader receives a request's schema pointer from the Dispatcher through a hardware queue and issues a memory request for that address to the Block Buffer. All cache blocks must be re-requested from the memory hierarchy the *first* time they are accessed in each transformation, because the Block Buffer is non-coherent. The Block Buffer returns a cache line containing schema fields, which the Reader stores in a dedicated Field Buffer. The Reader then fetches a field from the Field Buffer, extracts the data pointer, and issues a read request to the Block Buffer. If a field is a sub-message, such as the `Phone` field of `Person` in Fig. 4, the Reader recursively fetches the schema of that sub-message in a depth-first manner.

The Block Buffer returns a cache line containing the field's raw data, which the Reader stores in the Data Buffer. The Reader then extracts the required data (in *Chunks*) from the Data Buffer based on the field's type, and forwards it to the Converter to carry out the transformation. The Reader also calculates the offset where the Writer should place the transformed data, again depending on the schema. To illustrate this process, consider the `string` field in Figure 4. Once the second field of the schema is present in the Field Buffer, the Reader determines that it is a `string` of length eight. The Reader picks the correct eight bytes from the cache line, forms a Chunk with the correct output buffer offset and transformation type, and sends it to the Converter.

During deserialization, the Reader fetches data from the serialized buffer, finds the corresponding field in the schema, and passes the information to the Converter.

4.4 Converter

The Converter takes in the Chunks sent by the Reader and performs the required data transformation. The Chunks contain information that identifies the field's type and therefore what operation to execute. A small (128-entry) instruction memory stores a sequence of instructions for each application-defined type to perform the conversion. This memory is initialized when the application requests to use the accelerator (see §3.2) and is indexed by the type field in the Chunk. After data is transformed, the Converter passes the converted bytes to the Writer to be written to the output buffer.

The Converter is implemented as a simple pipeline with the following four stages: instruction fetch, decode and register file read, execute, and register file writeback. The field type included in the Chunk indicates the entry in the instruction memory that the Converter executes. For common data types (e.g., `varint`), a single instruction performs the conversion. Other transformations that do not have specialized instructions can execute a sequence of instructions at the cost of reduced throughput.

4.5 Writer

The Writer receives transformed data from the Converter, and writes it at the appropriate location in the output buffer, which is identified by a `(base,offset)` pair. The base address is supplied by the requesting core, and is passed to the Writer by the Dispatcher, while the offset is calculated by the Reader and passed to the Writer. The Writer contains internal write buffers that assemble a cache line of transformed data from the Converter, and writes it through the Block Buffer to the on-chip memory hierarchy. During deserialization, the Writer also writes the data pointers in the schema. Finally, once the Writer issues all the writes for a request, it notifies the Dispatcher of completion.

4.6 Transformation Pipeline Abstraction

A *Transformation Pipeline* is architected as a decoupled access-execute pipeline [43], and includes a single Reader, Converter, and Writer. We provision a single component per pipeline because when operating at peak throughput, a Reader can produce one Chunk per cycle if it is picking bytes from a contiguous array. A single Converter and Writer can keep up with this peak throughput. Once the Reader has finished queuing all of the Chunks for a message, it can continue to the next message while the Converter and Writer complete the transformations and writebacks. We next identify two key optimizations to improve pipeline throughput.

Prefetching: Each Transformation Pipeline's throughput heavily depends on memory access latency. Before any Converter can begin transforming data, a Reader must perform at least two memory accesses, one the for schema and another for the corresponding data. More accesses are required for sub-messages. However, as the Reader has access to the message schema in its Field Buffer, it can issue prefetches for each upcoming field and overlap the access latency. The Reader's prefetches attain 100% accuracy because the message's schema explicitly contains the address of each field.

Time-Sharing: Even with prefetching, we find that the pipeline still spends the majority of its cycles waiting for memory accesses. To increase utilization further, the pipeline can be time-shared among multiple requests. This technique is similar to coarse-grained multithreading in CPUs [35] and requires keeping multiple request contexts per Reader, which can be rotated in one cycle. The Converter and the Writer do not require contexts as they do not retain message state. Time-sharing provides almost the same performance as physically replicating the entire pipeline. The optimal degree of time-sharing is limited by the pipeline's idle fraction. For example, a pipeline that is stalled 75% of the time will have a utilization of 100% with four contexts. Adding contexts beyond this point will only increase request latency.

Table 1. Architectural simulation parameters.

Cores	ARM Cortex-A57-like; 64-bit, 2GHz, OoO 3-wide dispatch/retirement, 128-entry ROB, TSO
L1 Caches	32KB 2-way L1d, 48KB 3-way L1i, 64-byte blocks 2 ports, 32 MSHRs, 2-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 8MB total 16-way, 1 bank/tile 6-cycle access
Coherence	Directory-based Non-Inclusive MESI
Interconnect	2D mesh, 16B links, 3 cycles/hop
Memory	45ns access latency
OP	Block Buffer: 8KB, 2-way, 64-byte blocks, LRU 64MSHRs, 1 cycle hit, 2 read/write ports TLB: 2MB pages, 64 entries, direct mapped

5 Methodology

CPU Performance Study. To study the performance of DT on real hardware, we used an X-Gene system-on-chip with eight ARM Atlas A57 cores available on CloudLab [49]. Our microbenchmark runs (de)serialization tasks using Google Protobuf v3.7 and the message types in Table 2. We use the C API for Unix time and `perf` to measure transformation throughput and dynamic instruction count respectively.

System Organization. We simulate a 16-core ARMv8 server running Ubuntu Linux 18.04 in full-system cycle-level detail, by combining the QEMU emulator with the timing models from the Flexus simulator [50]. Table 1 summarizes the simulation parameters. All workloads are pinned on 15 cores, leaving one core for OS threads and interrupts. OP is attached to a corner tile of the NoC mesh, which has access to two NoC links. Therefore, OP has a total read/write bandwidth of 32 bytes/cycle i.e., 512Gbps.

Table 2. Message types and their characteristics.

Message Type	R/W Ratio	Max Depth	Size (B)
Flat	2.6	1	485
Mixed	2.75	2	297
Nested	4.25	2	232

Microbenchmark. We use a multithreaded microbenchmark that generates (de)serialization tasks based on Google Protobuf. In order to directly evaluate the maximum throughput of OP, the microbenchmark sends (de)serialization requests to OP in a tight loop. This scenario represents the upper bound of the load offered to OP, as in a real-world deployment, the application will also consume CPU time. As specified in §3.2, each core sends a transformation request to OP with MMIO writes, and repeatedly polls the address to check if the request is complete. Once the request completes, the core generates a new request and sends it to OP.

To choose representative messages to be transformed, we create three message classes shown in Table 2, based on prior work [44]. The sizes of these messages are chosen to represent the fact that the majority of network packets sent by latency-critical applications are sub-1KB [33, 41]. The R/W ratio is the number of bytes that must be read for each byte written in the serialized output, and depends on each field's depth and type. For example, `varints` are converted to different byte-streams based on their values, where `strings` have a R/W ratio of one. Moreover, the depth of each field (i.e., the number of sub-messages that must be parsed before returning to the top level) increases the read/write ratio. We analyze these effects in §6.1.

Microservices. We also evaluate OP using three microservices taken from DeathStarBench [14], which use Apache Thrift's RPC stack. Each microservice runs in isolation on the CPU, and uses Thrift's in-memory transport layer. We measure the latency of each service starting from the point the application receives a request until it finishes processing it and sends out the response. This includes RPC processing, message (de)serialization and the actual service. The microservices perform URL-shortening, user login, and read-post operations. Our experiment compares each microservice's latency between a completely CPU-centric deployment, and one where (de)serialization uses OP.

Synthesis. To estimate OP's area and power, we implemented OP in VHDL and synthesized it with the Synopsys Design Compiler [47] using TSMC 28nm technology (Core library: TCBN28HPMBWP35, Vdd: 0.9V). We use a 2GHz clock rate and set the compiler to the high area optimization target. The synthesized RTL only takes into account the Dispatcher, Reader, Converter and Writer. We add the power and area of the Block Buffer and TLB using CACTI 6.5 [34]. Finally, we compare our area and power overheads with Cortex-A57 numbers from prior work [38] in Table 4.

Notation for OP Configurations. To aid explanation of OP's possible configurations, we introduce the following notation: $OP_{\{i,j\}}$ refers to OP with i physical transformation pipelines, with each being time-shared between j messages.

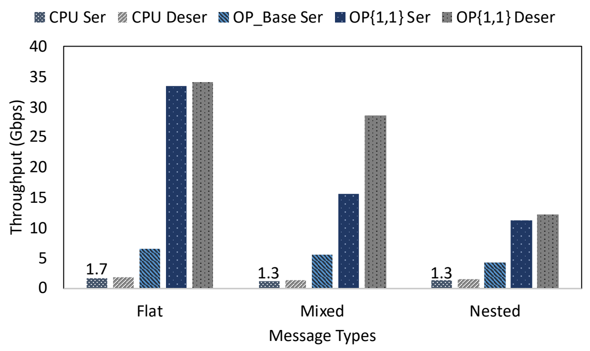


Figure 7. Data transformation throughput comparison of a single core with $OP_{\{1,1\}}$.

6 Evaluation

Our evaluation focuses on Optimus Prime's ability to transform data at the bandwidths of modern NICs. We first evaluate the benefits of OP using a single pipeline. We then evaluate an OP configuration which exploits parallel pipelines to match the NIC bandwidth. Next, we evaluate the impact of time-sharing and its effectiveness in improving pipeline utilization, thus reducing the number of required physical pipelines. Finally, we evaluate three microservices using OP to show the reduction in service latency, and conclude by analyzing the power and area of our synthesized RTL.

6.1 Single-Pipeline OP Throughput

We first measure the performance of OP configured with a single physical pipeline ($OP_{\{1,1\}}$) against CPU-centric DT, and plot the results in Fig. 7 for all three message classes. To isolate the improvement from Converter specialization, we also measure a configuration labeled *OP_Base Ser* that disables pipelining and prefetching. Fig. 7 shows that a CPU core can at best achieve a throughput of ~1.7Gbps for serialization, while OP is ~5× faster. *OP_Base Ser*'s throughput is limited because it spends most of its time waiting for data from the memory hierarchy.

Next we enable pipelining and prefetching; the throughput of this configuration is shown by the $OP_{\{1,1\}}$ Ser and $OP_{\{1,1\}}$ Deser bars. Prefetching and pipelining overlap the latency of transformations and memory accesses, improving throughput by another 2-4×. The key enabler for this overlap is our schema, which represents each field as a {type, address} pair, allowing OP to extract field-level parallelism. Such parallelism allows $OP_{\{1,1\}}$ to reduce the average field read latency from 27 cycles to 9 with prefetching. The CPU baseline does not attain this parallelism because the transformations are compiled into serial instruction slices that are lengthy, and highly control- and data-dependent.

The Nested message class represents the worst case performance for OP with a throughput of ~11Gbps, because every field in this message class is a sub-message. Therefore, each schema field must be read before the sub-message's data can be forwarded to the Converters. Additionally, the prefetcher only operates at the top message level, and therefore it does not overlap accesses further than the schemata of the first level of sub-messages. The Flat message class exhibits ~33Gbps of throughput, because all the schemata and data can be prefetched in parallel. Mixed messages have characteristics of both flat and nested, with OP reaching ~15Gbps.

Deserialization exhibits higher throughput because OP reads already-serialized items from a contiguous buffer, rather than making dependent accesses to the data elements, thus enjoying high spatial locality. However for messages which exhibit more nested fields, the degree of dependent accesses to the schema grows and limits throughput. The bottleneck in $OP_{\{1,1\}}$ is the serial processing of messages by a single transformation pipeline. Given that messages are naturally independent of each other, we now evaluate configurations with multiple pipelines. For brevity's sake, all further experiments only display results for serialization as deserialization has similar performance.

6.2 Parallel-Pipeline OP Throughput

Although $OP_{\{1,1\}}$ attains 9-20× higher serialization throughput than a core, there is significant headroom left to attain the 40Gbps sustainable by modern NICs. Next, we measure a scale-up OP by adding transformation pipelines which operate in parallel. Fig. 8 depicts the serialization throughput for $OP_{\{n,1\}}$ as we vary the number of transformation pipelines.

Flat messages achieve 40Gbps with only two pipelines, benefiting the most because they have the fewest dependent memory accesses. In contrast, Nested messages require four pipelines to achieve 40Gbps, and Mixed messages require three. Throughput increases linearly with up to three pipelines in all cases, because each extra pipeline adds additional independent memory accesses and transformations. Overall, our OP design can easily meet the target NIC bandwidth of 40Gbps for all the three message classes.

The throughput plateaus beyond a certain number of pipelines because they, in aggregate, exhaust the available NoC bandwidth. Flat messages are the most read-efficient (lowest R/W ratio) class of messages, and therefore generate less NoC traffic and higher throughput. In contrast, Nested messages require more reads per write (i.e., they have a greater R/W ratio), thus limiting the OP's serialization throughput to ~50Gbps. We confirm that each configuration has reached the maximum link bandwidth of 512Gbps by summing the bandwidth needed for the schema and message data, the additional NoC header overhead, and other on-chip coherence protocol requests.

When OP saturates the NoC links of the tile it is attached to, the whole NoC has an average link utilization of 16%. Preserving the NoC bandwidth that is available to the core on

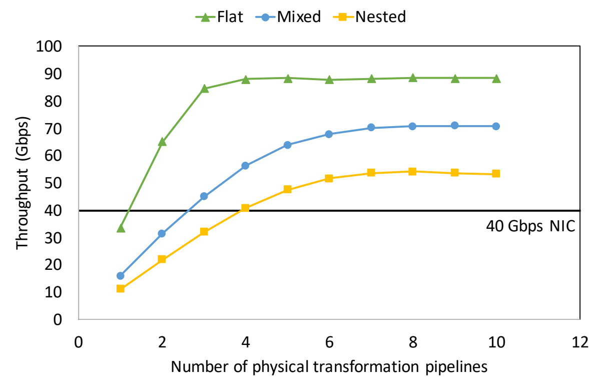


Figure 8. Serialization throughput with $OP_{\{n,1\}}$.

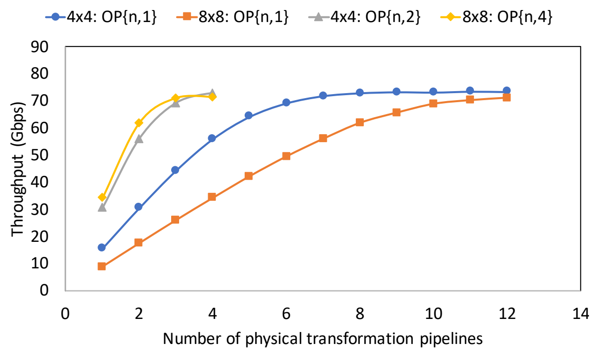
the contended link would require slightly over-provisioning the NoC's link width. The silicon costs of doing so are negligible, as the per-tile cost of the NoC components has been shown to be less than 1.5% [32].

6.3 Time-Shared Pipeline OP

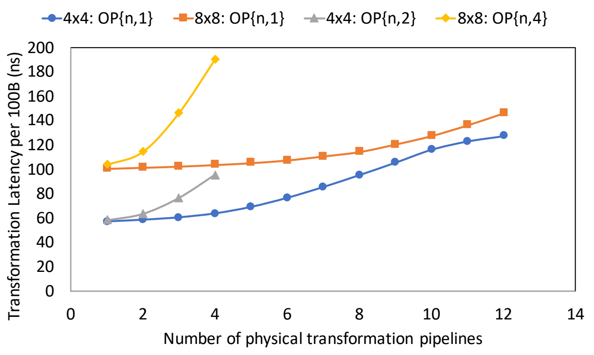
This section quantifies the benefits of time-sharing transformation pipelines and studies the effects of a larger diameter NoC on OP's performance. Fig. 9 illustrates the impact of longer average memory access latency (AMAT) on the number of pipelines required to attain peak throughput, by plotting $OP_{\{n,1\}}$'s throughput and latency per $100B$ for Mixed messages for a 4×4 and an 8×8 mesh. The 8×8 mesh has twice the AMAT of the 4×4 mesh, resulting in half the throughput for an equivalent OP configuration. Doubling the number of pipelines for 8×8 mesh OP recovers the original throughput of the 4×4. Following this trend, while the NoC link attached to OP saturates with six pipelines in the case of the 4×4 mesh, we need 12 pipelines to saturate the same link in an 8×8. Beyond this point, increasing the number of pipelines results in elevated latency due to contention for NoC bandwidth.

When time-sharing is enabled, OP can continue to issue memory accesses to hide cycles where the pipeline is idle. For instance, with a time-sharing degree of two ($OP_{\{n,2\}}$) we are able to saturate OP's NoC link in the 4×4 mesh with three pipelines as opposed to six in Fig. 8. The 8×8 mesh benefits more from time sharing due to its larger average latency, and requires a time-sharing degree of four to saturate OP's NoC link with three physical pipelines. Fig. 9a shows that a time-sharing degree of four, $OP_{\{n,4\}}$, has a nearly identical throughput curve with increasing n as the time-sharing degree of two for a 4×4 mesh. Time-sharing enables OP to achieve roughly the same throughput per pipeline even with different NoC sizes. As long as there is available NoC bandwidth and request-level parallelism, adding more pipelines or time-sharing pipelines increases attainable throughput.

Finally, in Table 3 we place the transformation latency achieved by OP in the context of future datacenters. We assume protocol processing latencies of ~850ns as claimed by *eRPC* [27]. Even with a large host node, $OP_{\{3,4\}}$ achieves average serialization latency of 430ns, which is less than the



(a) Transformation bandwidth.



(b) Average Transformation latency per 100B.

Figure 9. OP throughput and latency for serialization over Mixed messages comparing different NoC sizes.

Table 3. Communication latency breakdown (in μs) into Wire Time (WT), Protocol Processing (PP), and Data Transformation (DT). $OP_{\{3,2\}}$ sits in a 4×4, and $OP_{\{3,4\}}$ in an 8×8 mesh.

Network Stack	WT	PP	DT
40 Gbps + TCP	0.05	**20**	4
40 Gbps + eRPC	0.05	0.85	**4**
40 Gbps + eRPC + $OP_{\{3,2\}}$	0.05	**0.85**	0.22
40 Gbps + eRPC + $OP_{\{3,4\}}$	0.05	**0.85**	0.43

time spent in the eRPC stack. We conclude that OP removes DT from the critical path of inter-microservice communication, as it achieves network line-rate and latency below that of the current-best networking protocols.

Scale-Out OP: As shown in Fig. 8, improving throughput by scaling the number of transformation pipelines is constrained by the bandwidth of the connected NoC links. However, with the roadmap for Ethernet and Infiniband NICs already forecasting speeds as high as 1Tbps [24, 48], OP needs to be scaled out and attached to multiple NoC links. This is possible by replicating the whole accelerator across multiple tiles of the NoC, providing OP more aggregate NoC bandwidth and allowing DT throughput to meet NIC bandwidth.

6.4 Case Study on Microservices

In order to quantify OP's benefits on the performance of real microservices, we run three microservices taken from DeathStarBench [14]. Fig. 10 breaks down the total latency of each microservice into the time spent in the application, Thrift's data transformation and the rest of the Thrift RPC stack. Compared to our CPU baseline, OP reduces DT latency by up to 10× and service latency by up to 30%. Our improvements are upper-bounded by the fraction of the baseline service time spent in data transformation.

The highest improvement is registered on the URL-Shorten benchmark, where initially ~31% of the time was spent on data transformation. Even though this benchmark uses small

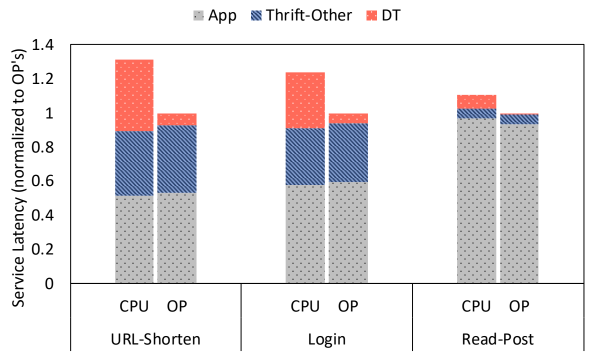


Figure 10. Microservices: CPU vs. OP.

incoming and outgoing messages, OP still decreases DT latency by 5.65×, benefiting from the specialized Converters. Similarly, OP gives 24% improvements for the Login service. The smallest improvement is obtained for the Read-Post microservice because only 7% of the baseline service's time is spent on DT. Full service deployments commonly include hundreds of sequentially connected microservices [9, 14, 31, 45, 51]; in such a case, the cumulative latency improvements from OP will be significantly greater.

6.5 Synthesis Results

To model the area and power consumption of OP, we synthesized our RTL design using TSMC 28nm technology, and display the results in Table 4. Configured with 12 pipelines, $OP_{\{12,1\}}$ requires $0.45mm^2$ of area, and consumes $532mW$ when operating at $2GHz$. Additionally, each time-shared pipeline requires an enhanced 4-way Reader to switch between different message contexts. Such time-shared Readers have ~20% greater area and power overhead compared to normal Readers. Fortunately, time-sharing reduces the number of transformation pipelines required at saturation from 12 to 3. Therefore the $OP_{\{3,4\}}$ configuration achieves the same performance as $OP_{\{12,1\}}$ with a silicon area reduction of 60% and a performance/watt improvement of 3.5×. Compared to a CPU core, $OP_{\{3,4\}}$ achieves 2075× higher performance/watt.

Table 4. Synthesis results for different configurations of OP, compared to the CPU baseline. All throughput numbers are for serializing Mixed messages on the 64-core setup, and all performance per watt numbers are normalized to the CPU.

	Power [mW]	Throughput [Gbps]	Area [mm^2]	Performance per Watt
CPU	5400	1.3	2.57	1
$OP_{\{1,1\}}$	58	8.8	0.12	655
$OP_{\{12,1\}}$	532	73	0.45	593
$OP_{\{3,4\}}$	152	73	0.19	2075

Finally, we compare the area of a shared version of OP to a core-private version, as discussed in §3.4. Private OPs do not require a Block Buffer and share the CPU core's TLB, and therefore require $0.03mm^2$ of area. The silicon area of $OP_{\{3,4\}}$ is only ~7% of a single CPU core, whereas having 64 private OPs, one for each core, costs ~75% of a core. This overhead from replication justifies our choice of having a shared OP component.

7 Related Work

Other Data Transformation Frameworks. We focus on Protobuf [19] and Thrift [3] as two state-of-the-art frameworks for generalized cross-language data transformation. Optimus Prime is designed to be a *general-purpose* DT accelerator, which is compatible with other frameworks provided they simply update their *setter* methods to create each message's schema. Custom frameworks such as Google's FlatBuffers [18] sacrifice interoperability to remove the need for data transformation for use-cases where large immutable messages are sent among many participants (e.g., for online games). In these systems, messages are effectively serialized in memory *during creation*, amortizing the cost of DT and eliminating costly deserialization of the whole message when only parts of it are accessed. However, this comes at the price of more costly object creation, object immutability, less flexibility and larger messages, preventing such frameworks from being a good fit for general-purpose inter-language RPCs. OP instead targets general-purpose DT frameworks designed for inter-microservice communication.

Architectures for Data Ingestion & Streaming. Intel has recently released the specifications for an integrated Data Streaming Accelerator (DSA), supporting operations like data copying, virtual switching, and integrity checking [25]. Although DSA does not currently target general-purpose DT, our key insights would equally apply to DSA, as it follows many of our design choices, e.g., integration into to the virtual memory system. Applying our transformation schema would enable DSA's internal hardware to unlock the field-level parallelism inherent in transformation tasks. SoC designers wishing to perform general-purpose DT with DSA could construct specialized DSA Engines (which are in principle similar to our Converters) for common DT tasks.

Finite automata (FA) processing is an emerging computational model that promises orders of magnitude better performance than CPUs in executing Finite State Machines (FSMs) [11, 16, 46]. FSMs traditionally suffer from complex control flow, limiting the benefits of branch prediction, and irregular memory access patterns, reducing the effectiveness of caching. We observe similar behavior for code generated from DT frameworks. Therefore, FA accelerators for tasks such as pattern matching or replacement could easily be deployed as a type of Converter in OP's pipelines. UDP [12] applies the FA model to a coarse-grained class of workloads such as data mining and CSV file parsing. Their architecture is targeted towards bulk loading and cleaning of batches of data, and motivate UDP by comparing CPU processing time to disk I/O. In contrast, our work targets eliminating DT as a bottleneck for latency-critical inter-microservice RPCs.

ISA Extensions for DT. CPU vendors have also realized the difficulties in expressing transformations using existing ISAs. In fact, Intel has already been granted a patent for ISA extensions to x86-64 which provide dedicated support for specific DT operations [20]. The performance impact of ISA extensions would be more or less similar to our specialized Converters, which are specialized pipelines tailored for transformation. However, the serialization of an entire message is still expressed as a serial sequence of many transformation instructions with implicit parallelism. Our work goes further by proposing an entire new abstraction for explicit parallelism between the many fields of a message.

8 Conclusion

With improvements in network technology and protocol processing, data transformation forms a significant portion of end-to-end communication latency. We propose an accelerator, Optimus Prime (OP), to transform data at network line-rate, thus removing data transformation from the critical path. Our key contributions are a parallel schema abstraction that allows our hardware implementation to (de)serialize all the fields of a message in parallel, and the hardware accelerator design to efficiently use the parallel schema. OP achieves ~60× higher throughput and 2075× better performance per watt than traditional CPU cores. It also shortens the service latency of our evaluated microservices by up to 30%.

Acknowledgments

We thank the anonymous reviewers and Alexandros Daglis for their precious comments and feedback. This work was partially funded by SNSF's *Memory-Centric Server Architecture for Datacenters* project, Microsoft Research's joint research agreement for *Near-Memory Data Services*, and Oracle's research donation for the *Accelerating Distributed Systems with Advanced One-Sided Operations* project.

References

[1] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[2] Amazon Web Services, Inc. [n.d.]. *Amazon EC2 Instance Types*. Retrieved November 29, 2018 from https://aws.amazon.com/ec2/instance-types/

[3] Apache Software Foundation. [n.d.]. *Thrift*. Retrieved August 16, 2019 from https://thrift.apache.org/

[4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers.

[5] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*. 49–65.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*. 49–60.

[8] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. FBOSS: building switch software at scale. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 342–356.

[9] Adrian Cockcroft. 2015. Microservices the Good Bad and the Ugly. Retrieved August 16, 2019 from https://www.slideshare.net/adriancockcroft/microservices-the-good-bad-and-the-ugly

[10] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 33–48.

[11] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 12 (2014), 3088–3098.

[12] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. 2017. UDP: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 55–68.

[13] Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. *Computer Architecture Letters* 17, 2 (2018), 155–158. https://doi.org/10.1109/LCA.2018.2839189

[14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*. 3–18.

[15] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*. 249–264.

[16] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 44:1–44:12.

[17] Google. [n.d.]. *C++ Arena Allocation Guide* . Retrieved April 5, 2019 from https://developers.google.com/protocol-buffers/docs/reference/arenas

[18] Google. [n.d.]. *FlatBuffers*. Retrieved April 5, 2019 from https://google.github.io/flatbuffers/

[19] Google. [n.d.]. *Protocol Buffers*. Retrieved November 30, 2018 from https://developers.google.com/protocol-buffers/

[20] James D. Guilford and Vinodh Gopal. 2016. Instruction Set for Variable Length Integer Coding. https://patents.google.com/patent/US20180095760A1/en

[21] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 29–42.

[22] Todd Hoff. 2016. Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories. Retrieved August 16, 2019 from http://highscalability.com/blog/2016/10/12/lessons-learned-from-scaling-uber-to-2000-engineers-1000-ser.html

[23] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito IV, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. 2017. SVE: Distributed Video Processing at Facebook Scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 87–103.

[24] Infiniband Trade Association. 2018. *Infiniband Roadmap*. Retrieved November 29, 2018 from https://www.infinibandta.org/infiniband-roadmap/

[25] Intel Corp. 2019. Intel Data Streaming Accelerator Preliminary Architecture Specification. Retrieved January 16, 2020 from https://software.intel.com/en-us/download/intel-data-streaming-accelerator-preliminary-architecture-specification

[26] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the 2018 EuroSys Conference*. 33:1–33:15.

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. 1–16.

[28] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2016. Profiling a Warehouse-Scale Computer. *IEEE Micro* 36, 3 (2016), 54–59.

[29] Staci Kramer. 2011. The Biggest Thing Amazon Got Right: The Platform. Retrieved August 16, 2019 from https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform

[30] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-mei W. Hwu. 1995. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*. 138–150.

[31] Tony Mauro. 2015. Adopting Microservices at Netflix: Lessons for Architectural Design. Retrieved August 16, 2019 from https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices

[32] Michael McKeown, Alexey Lavrov, Mohammad Shahrad, Paul J. Jackson, Yaosheng Fu, Jonathan Balkind, Tri M. Nguyen, Katie Lim, Yanqi Zhou, and David Wentzlaff. 2018. Power and Energy Characterization of an Open Source 25-Core Manycore Processor. In *Proceedings of the 24th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 762–775.

[33] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 221–235.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 3–14.

[35] Mario Nemirovsky and Dean M. Tullsen. 2013. *Multithreading Architecture*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00458ED1V01Y201212CAC021

[36] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 327–341.

[37] Fabian Ohler, Markus C. Beutel, Sevket Gökay, Christian Samsel, and Karl-Heinz Krempels. 2018. A Structured Approach to Support Collaborative Design, Specification and Documentation of Communication Protocols. In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018*. 367–375. https://doi.org/10.5220/0006787503670375

[38] Ali Pahlevan, Javier Picorel, Arash Pourhabibi Zarandi, Davide Rossi, Marina Zapater, Andrea Bartolini, Pablo García Del Valle, David Atienza, Luca Benini, and Babak Falsafi. 2016. Towards near-threshold server processors. In *Proceedings of the 2016 Design, Automation, and Test in Europe Conference and Exhibition (DATE)*. 7–12.

[39] Dionisios N. Pnevmatikatos and Gurindar S. Sohi. 1994. Guarded Executing and Branch Prediction in Dynamic ILP Processors. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*. 120–129.

[40] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 325–341.

[41] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*. 123–137.

[42] A. Schaffer. 2018. Testing of microservices. Retrieved August 16, 2019 from https://labs.spotify.com/2018/01/11/testing-of-microservices

[43] James E. Smith. 1984. Decoupled Access/Execute Computer Architectures. *ACM Trans. Comput. Syst.* 2, 4 (1984), 289–308.

[44] Akshitha Sriraman and Thomas F. Wenisch. 2018. μ Suite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*. 1–12. https://doi.org/10.1109/IISWC.2018.8573515

[45] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 177–194.

[46] Arun Subramaniyan and Reetuparna Das. 2017. Parallel Automata Processor. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 600–612.

[47] Synopsys. [n.d.]. *Synopsys Design Compiler*. Retrieved November 29, 2018 from https://synopsys.com

[48] The Ethernet Alliance. 2018. *The 2018 Ethernet Alliance Roadmap*. Retrieved November 29, 2018 from https://ethernetalliance.org/the-2018-ethernet-roadmap/

[49] The University of Utah. [n.d.]. CloudLab Hardware. https://www.cloudlab.us/hardware.php Retrieved 15-Jan-2020.

[50] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.

[51] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018,Carlsbad, CA, USA, October 11-13, 2018*. 149–161. https://doi.org/10.1145/3267809.3267823