Profiling Hyperscale Big Data Processing

Abraham Gonzalez
abe.gonzalez@berkeley.edu
Google, UC Berkeley
Berkeley, CA, USA

Aasheesh Kolli
aasheesh@google.com
Google
Mountain View, CA, USA

Samira Khan
samirakhan@google.com
Google
Mountain View, CA, USA

Sihang Liu*
sihangliu@uwaterloo.ca
University of Waterloo
Waterloo, ON, CAN

Vidushi Dadu
vidushid@google.com
Google
Mountain View, CA, USA

Sagar Karandikar
sagark@eecs.berkeley.edu
UC Berkeley, Google
Berkeley, CA, USA

Jichuan Chang
jichuan@google.com
Google
Mountain View, CA, USA

Krste Asanović
krste@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Parthasarathy Ranganathan
partha.ranganathan@google.com
Google
Mountain View, CA, USA

ABSTRACT

Computing demand continues to grow exponentially, largely driven by "big data" processing on hyperscale data stores. At the same time, the slowdown in Moore's law is leading the industry to embrace custom computing in large-scale systems. Taken together, these trends motivate the need to characterize live production traffic on these large data processing platforms and understand the opportunity of acceleration at scale.

This paper addresses this key need. We characterize three important production distributed database and data analytics platforms at Google to identify key hardware acceleration opportunities and perform a comprehensive limits study to understand the trade-offs among various hardware acceleration strategies.

We observe that hyperscale data processing platforms spend significant time on distributed storage and other remote work across distributed workers. Therefore, optimizing storage and remote work in addition to compute acceleration is critical for these platforms. We present a detailed breakdown of the compute-intensive functions in these platforms and identify dominant key data operations related to datacenter and systems taxes. We observe that no single accelerator can provide a significant benefit but collectively, a *sea of accelerators*, can accelerate many of these smaller platform-specific functions. We demonstrate the potential gains of the sea of accelerators proposal in a limits study and analytical model. We perform a comprehensive study to understand the trade-offs between accelerator location (on-chip/off-chip) and invocation model (synchronous/asynchronous). We propose and evaluate a *chained accelerator* execution model where identified compute-intensive functions are accelerated and pipelined to avoid invocation from the core, achieving a 3x improvement over the baseline system while nearly matching identical performance to an ideal fully asynchronous execution model.

*Work done while at Google.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589082

CCS CONCEPTS

• **Information systems → Database query processing**; • **Computer systems organization → Cloud computing**.

KEYWORDS

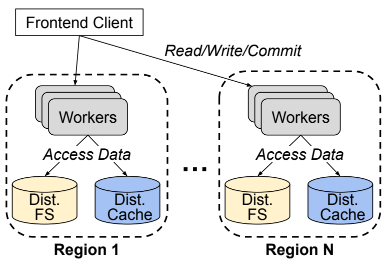
data analytics, databases, hyperscale computing, cloud computing, warehouse-scale computing, profiling, accelerators, accelerator-chaining

ACM Reference Format:
Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanović, and Parthasarathy Ranganathan. 2023. Profiling Hyperscale Big Data Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3579371.3589082

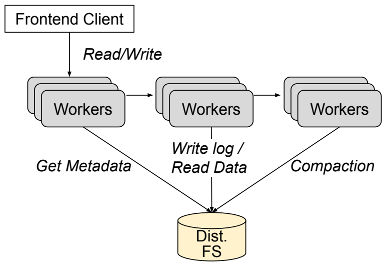
1 INTRODUCTION

Growing volumes of data are causing demand for computing to increase at phenomenal rates. More than 2.5 quintillion bytes of data are expected to be generated per day throughout the world [35]. The global "data sphere" is forecast to be greater than 175 zettabytes and commercial big data solutions are growing in capacity and features to satisfy this demand [3]. This growth of data-centric computing is exemplified by the growing computing needed for large databases, data warehouses, and data lakes in hyperscaler and cloud companies (e.g., Google, Amazon, Microsoft, and Meta).

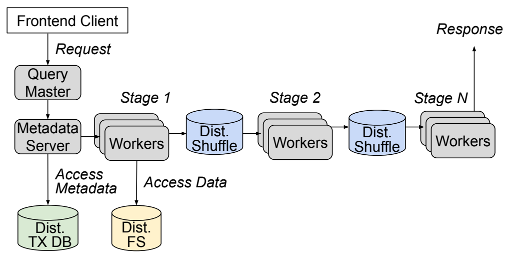
At the same time, Moore's law is slowing down, stressing traditional assumptions around cheaper and faster systems every year. This trend has led to new solutions that consider the entire data center as a computer (warehouse-scale computing [28, 56]) as well as innovative new custom silicon [5, 7, 27, 30, 45]. Underpinning both approaches is a deep understanding of key workload behavior at scale, allowing for more vertically integrated system designs.



(a) Spanner



(b) BigTable



(c) BigQuery

Figure 1: Google Big Data Processing Architectures

However, despite several prior studies on data processing platforms [19, 28, 40, 44, 54–56, 66], there is a lack of research and deep quantitative information on *hyperscale big data processing platforms*, including their behavior on real-world production traffic and corresponding opportunities for hardware acceleration.

Addressing this, we present, to the best of our knowledge, the first large-scale profiling study of hyperscale big data processing platforms at Google. We contribute the following:

- We *characterize* three key types of production big data processing platforms serving live traffic: *a distributed SQL and NoSQL database (Spanner and BigTable), and a distributed data analytics query engine (BigQuery)*. We present an end-to-end execution breakdown of time spent on compute, distributed storage, and other remote work, such as shuffle and compaction. Our characterization identifies the need for software-hardware co-design, as 52% of end-to-end time is spent on remote work and distributed storage operations.
- We provide *detailed per-platform workload-level breakdowns* and identify key compute-intensive hardware acceleration targets. We highlight that over 72% of time is spent on *datacenter and system tax* components, an important unique feature of *distributed* big data processing. With no individual function bottleneck, we argue that collectively, a *sea of accelerators*, can accelerate many of these smaller functions along with datacenter and system tax operations.
- We perform *an accelerator limit study and trade-off analysis with an analytical model* for various sea of accelerators design points, varying placement (on-chip/off-chip), and the amount of overlap in execution (synchronous/asynchronous acceleration). Our results show that co-designs eliminating storage and remote work overheads provide more significant benefit. Additionally, with large working sets, analytics platforms can slow down due to high off-chip accelerator data transfer costs. The most benefit is achieved with asynchronous accelerator execution where all accelerator invocations are parallelized.
- Based on our characterization and analysis, we propose a *chained accelerator execution model*, where consecutive operations are sent to the next accelerator without core coordination, that achieves less than a 1% difference compared to an ideal fully asynchronous execution model. Through validation with a synthetic benchmark that computes a SHA3 hash of fleet-wide representative protobuf messages, our chained model obtains a 6.1% difference compared to an open-source accelerated RISC-V system-on-chip baseline.

2 GOOGLE BIG DATA PROCESSING

This section first compares smaller-scale research systems with production hyperscale big data processing systems. We then describe the main big data processing systems at Google and outline the goals of the paper.

2.1 Characteristics of Production Systems

Traditionally, databases started as a single node system where requests and responses are served from a local storage. With the growth of datasets, horizontal scaling techniques, such as sharding or data partitioning, were often used to distribute data between small node clusters. Such systems include [2, 12, 25, 37, 39, 61]. Concurrently, the growing volumes of data in hyperscale deployments pushed the industry towards an extreme horizontal scaling approach.

In the case of Google, data processing systems are highly distributed and serverless similar to other hyperscale platforms [4, 11, 13, 20, 41, 58]. Incoming requests can be handled by many homogeneous modern Intel Xeon and AMD EPYC servers controlled by a cluster manager for resource provisioning and separated by a proprietary high-speed custom network [53, 59]. Working sets are often hundreds of petabytes or more and are managed through a distributed file system and caching layer, which partitions, replicates, and stores the data [15]. Finally, with data stored in different formats and locations, platforms are split into databases and query engines, helping to decouple data management from query execution. Examples of this trend are now emerging in academia, for example, with the use of Spark and Hadoop for SQL execution while matching some of the scale-out properties of industrial systems [9, 60].

2.2 Big Data Processing Overview

Figure 1 presents architecture overviews of three big data processing platforms at Google: two distributed databases – Spanner and BigTable – and one data analytics query engine, BigQuery. We chose these three processing platforms because they take a significant amount of fleet-wide CPU cycles (more than 10%), with over 90% of these cycles being high-priority production tier cycles. Additionally, they were chosen because they are highly optimized and tuned

Table 1: Storage-to-Storage Ratios. Petabytes of RAM to SSD to HDD owned per platform.

Storage-to-Storage Ratios RAM PiB : SSD PiB : HDD PiB		
Spanner	**BigTable**	**BigQuery**
1 : 8 : 90	1 : 16 : 164	1 : 7 : 777

over many engineer years for serving live production traffic from multiple internal and external consumers. These platforms ingest various data sources, from user-owned structured data to logging data obtained from monitoring. The distributed databases, Spanner and BigTable, offer users a traditional SQL and a NoSQL key-value store interface, respectively. BigQuery is used in combination with the databases to provide insights to downstream users and systems through SQL queries. We next describe these platforms in more detail.

2.2.1 Spanner. Spanner is a scalable, globally distributed, synchronously replicated database [15]. Figure 1a shows the high-level architecture of Spanner where workers are distributed globally between different regions and access data and metadata through the distributed caching and file system layers. Spanner supports both general-purpose transactions and SQL queries. Additionally, it supports sequentially consistent reads and writes while providing globally-consistent reads across the database. Databases built on Spanner can scale to petabytes in size. Users of Spanner include Advertising, Docs, Play, Photos, and a long tail of smaller applications.

2.2.2 BigTable. BigTable is a scalable, cluster-level key-value storage system [13]. Unlike Spanner, BigTable supports a loose set of consistency requirements for simple transactional queries. Similar to Spanner, it is also designed to scale to petabyte-sized databases and handles millions of requests per second. Figure 1b shows BigTable's architecture, where a single BigTable cluster stores a database table in multiple servers. Users of BigTable include Finance, Earth, Search, and many smaller applications.

2.2.3 BigQuery. BigQuery is a large-scale distributed multi-tenant query engine and data warehouse used for interactive data analysis in Google's production and cloud environments [14]. Unlike a pure MapReduce-like system, BigQuery provides the ability to have performant real-time interactive results (scan throughput of over a billion records per second) with structured SQL queries. Figure 1c shows the workflow of a query, where a series of intermediate servers process data and a distributed shuffle engine sends data to the next stage servers [36]. This platform has thousands of users running workloads such as analysis of crawled web documents, resolving issues from crash reports, and spam analysis.

2.3 Goals

Given the tremendous scale and complexity of the data processing platforms, this work aims to understand and characterize them from a systems and hardware perspective. We ask and answer the following questions in the rest of the paper.

- *Section 3:* How are these hyperscaler systems balanced when targeting extreme horizontal scaling? Is the storage to storage ratio keeping up with the growing demand?
- *Section 4:* These hyperscaler platforms run on thousands to millions of servers heavily, relying on distributed storage and inter-node communication. What is the main bottleneck in their end-to-end execution time and what kind of systems optimizations can help these platforms?
- *Section 5:* Where is the main bottleneck at each local node in these distributed platforms? Are there potential acceleration targets on those nodes?
- *Section 6:* What is the upper bound of software-hardware co-design for these platforms? What is an optimal model for the complexities of distributed and local node components in these platforms?

3 SYSTEM BALANCE

As data processed by big data platforms grows exponentially, we are interested in understanding the systems balance in these hyperscaler platforms. Table 1 presents storage-to-storage system balance ratios – a ratio of HDD, SSD, and RAM petabytes owned per platform – given by internal logging resources over a full week in 2022.

These platforms use large amounts of RAM for read caches and write buffers to minimize expensive accesses to disaggregated storage. For every 90, 164, or 777 bytes in HDD, a byte is allocated in RAM across Spanner, BigTable, and BigQuery, respectively. This high RAM usage makes these platforms expensive to operate. Disaggregated memory systems can potentially reduce these costs by allowing a peak-of-sum allocation versus a sum-of-peaks provisioning model [34] for large memory caches.

These data processing platforms also have large working sets that are too expensive to maintain entirely in memory. As a result, they typically employ SSD caches to minimize accesses to HDDs, as seen by the high RAM:SSD ratios. We observe that these platforms read from SSDs more frequently than from HDDs, suggesting that caching is an effective performance optimization. One promising approach is using machine learning to place data between the storage tiers [23, 38]. *The SSD to HDD ratio is quite high (approx. 10x to 110x) for the platforms*, suggesting that we have a unique opportunity to rethink the storage hierarchy and add more caching layers. Looking ahead, if storage were to grow at twice the rate of compute (e.g., domains like video), system balance across compute and storage will be further stressed, motivating rethinking memory and distributed storage hierarchies [45].

4 END-TO-END EXECUTION TIME BREAKDOWN

To accelerate a large distributed system, we first need to understand how time is spent within the system. This section presents an end-to-end execution breakdown of the three big data processing platforms and characterizes the time spent locally versus remotely in the distributed workflow.

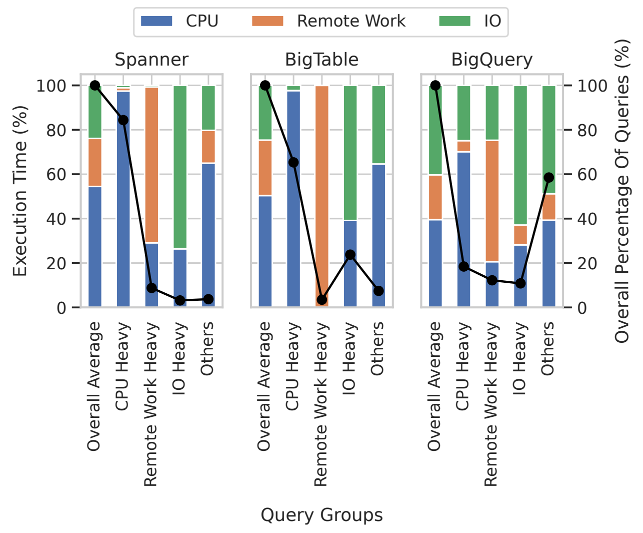


Figure 2: End-to-End Execution Time Breakdown. Execution time corresponds to the stacked bar portion of the graph, while percentage of queries corresponds to the line drawn.

4.1 Methodology

We profile Spanner and BigTable using Dapper, an internal RPC trace logging system that measures and traces RPCs between production services [52]. For BigQuery, we collect end-to-end time breakdowns from its internal timing logs. Given the large number of queries run in a day and the massive time spent on trace processing, we sample one-thousandth of all queries in a day for Spanner and BigTable. We categorized end-to-end execution time obtained into remote work, storage/IO, and CPU time. The remote work represents when a local server node is waiting for remote workers to complete operations such as consensus protocols for Spanner, compaction in remote storage for BigTable, and distributed shuffles for BigQuery. To match the RPC trace logs to BigQuery's timing logs, we categorized overlapped time first into remote work, then IO, then CPU time, assuming that CPU time was blocked on remote work and IO.

4.2 Time Breakdown

Figure 2 shows the end-to-end breakdown of the platforms. We separate the queries into five groups: "CPU Heavy", "IO Heavy", "Remote Work Heavy", "Others", and the "Overall Average" breakdown of all queries. We categorize CPU heavy queries as queries that spent more than 60% of time on CPU computation, and IO and Remote heavy queries as queries that spent more than 30% of the time on distributed storage and remote work. A few observations stand out.

Spanner and BigTable are primarily CPU heavy, while BigQuery has more IO and remote work: More than 60% of the queries are CPU heavy in Spanner and BigTable, where only 10% of the BigQuery queries are CPU heavy. Spanner and BigTable deploy better caching mechanisms for both data and metadata and therefore, most cycles are spent on performing computation. These two platforms are prime candidates for hardware acceleration. On the other hand, BigQuery, which mainly focuses on data analytics workloads, can benefit from better management of storage and remote work. BigQuery workloads are often larger and less cachable than the pure database workloads, for example, doing large scans over terabyte-sized tables. This breakdown matches our observation in Section 3 that data analytics engines can be more IO heavy than databases.

IO and remote work optimizations are important for overall system acceleration: Across all platforms, all queries spent 48%, 22%, and 30% of time on compute, remote work, and IO. As a result, 52% of end-to-end time is collectively spent on remote work and distributed storage operations. This indicates the importance of software-hardware optimizations, such as optimizing distributed shuffle mechanisms and consensus protocols over the network. Additionally, intelligently placing data closer to compute through new caching mechanisms or new memory tiers will also reduce data transfer over the network.

Table 2: Datacenter Tax Category Descriptions

Datacenter Tax	Description
Compression	(De)compression ops.
Cryptography	Hashing, security tools/infra., etc.
Data Movement	`mem{cpy,move}`, `copy_user` ops.
Mem. Allocation	Mem. reservation ops. (`malloc`, etc.)
Protobuf	(De)serialization setup and ops.
RPC	Remote procedure calls

5 CPU EXECUTION TIME BREAKDOWN

This section presents a breakdown of the CPU cycles spent on the three platforms isolated from non-CPU dependencies. We also study microarchitecture differences and identify optimizations based on these breakdowns.

5.1 Methodology

We use Google-Wide Profiling (GWP), a fleet-wide profiling tool, for sampling and collecting CPU profiles across machines from Google's production fleet over a single representative day in 2022 [28]. We manually categorize, prioritize, and aggregate returned samples by their leaf functions in the call stack. This allows us to introspect on CPU compute cycles and performance counters (e.g., branch misses per kilo-instruction) spent on specific functions to understand system bottlenecks.

5.2 Node-level Breakdown

To get a better intuition on common computing patterns and bottlenecks, we breakdown compute cycles into three broad categories: core compute, datacenter taxes, and system taxes.

- *Core compute is the essential business logic and core primitives of the specific data processing platform.* This category helps identify data processing patterns that are common across multiple platforms (e.g., joins and sorts).
- Datacenter taxes, shown in Table 2, are the key functions necessary to run hyperscale workloads [28, 56].

Table 3: System Tax Category Descriptions

System Tax	Description
EDAC	Error handing (checksums, etc.)
File Systems	IO backend client compute
Other Memory Ops.	Non-data-movement mem. ops.
Multithreading	Thread management overheads
Networking	Packet, web, server processing
Operating Systems	Kernel, syscalls, time ops.
STL	Standard fleet-wide libraries
Misc. System Taxes	Uncategorized ops.

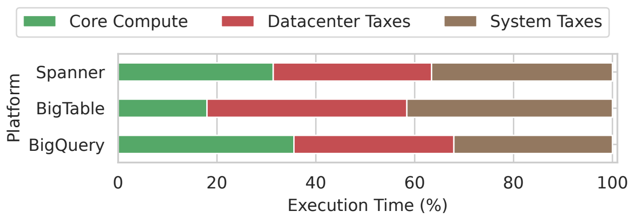


Figure 3: High-Level Application-Level Cycle Breakdown

Table 4: Spanner and BigTable Core Compute Descriptions

Operation	Description
Read	Read operations
Write	Write/commit operations
Compaction	Revision control/cleanup
Consensus	Replication and consensus protocols
Query	SQL-like compute
Misc.	Long-tail of labeled misc. compute
Uncategorized	Unlabeled compute

- *System taxes are overheads that are not considered traditional datacenter taxes but are shared amongst many production binaries.* Table 3 describes the overheads[1].

Figure 3 shows the compute cycles of each platform broken down into the three broad categories. The figure shows that *neither core compute, nor datacenter taxes, nor system taxes dominate overall compute cycles.* The *time spent on core compute operations specific to data processing is relatively small, 18% to 36% of total cycles.* 32% to 40% of CPU cycles are spent on datacenter taxes, while 32% to 42% of CPU cycles are attributed to system taxes. The relative fractions of core compute, datacenter tax, and system tax is a reflection of the local versus distributed design trade-offs in these systems. While traditional approaches to accelerating core compute database operators, such as aggregation and joining, can have some benefits, *all datacenter taxes, system taxes, and core compute need to be accelerated holistically to avoid diminishing end-to-end performance improvements, as shown by Amdahl's Law.*

[1]The "Other Memory Ops." category could be associated with datacenter taxes, but to stay consistent with the original definitions in [28] we assigned it to system taxes.

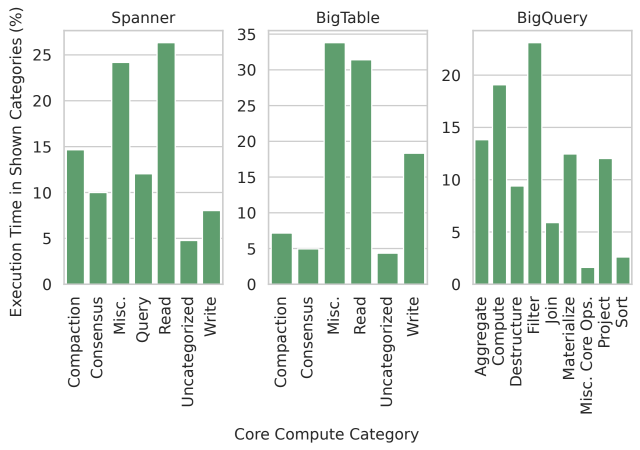


Figure 4: Core Compute Execution Breakdown

Table 5: BigQuery Core Compute Descriptions

Operation	Description
Aggregate	Compute/data-mov. for hash/sort aggs.
Compute	Col.-wise ops on pre-grouped aggs.
Destructure	Structured element field access
Filter	Scan/selection of rows
Join	Compute/data-mov. of hash/sort joins
Materialize	Construction of in-memory tables
Project	Retrieval of individual table columns
Sort	Non agg./join sort operations

5.3 Core Compute

Figure 4 shows the normalized breakdown of CPU cycles per fine-grained category within the platform core compute cycles. Tables 4 and 5 describe the individual categories.

We first observe that across all of the platforms, *no single fine-grained category dominates, indicating that there is no single-function accelerator that can dramatically improve performance.* However, there are fine-grained clusters of related functionality that can be combined to form *groupings of hardware accelerators* that can provide coverage of a large portion of cycles. For example, BigQuery spends the majority of its core compute cycles on functions such as filtering, aggregation, and compute (14% to 23%) once the data is retrieved from the underlying storage service or database. These functions can form the basis of a common set of hardware accelerators optimized for these operations instead of a single accelerator. Similarly, the databases (Spanner and BigTable) spend the majority of their cycles on read, write, and consensus protocols that could be accelerated together.

We also see the different acceleration candidates across these two classes of platforms based on the design choices made. BigQuery has fewer core compute read/write-like operations (low materialize and project categories). This is because it executes these operations as part of the datacenter/system tax categories when retrieving data from other backend services, without needing to enforce additional read/write semantics. In contrast, the databases devote large amounts of additional compute to ensure transaction semantics.

Finally, our study suggests that *clustering smaller cross-category accelerators together into a data processing shared accelerator complex can provide significant acceleration for hyperscale data processing.* This differs from prior works that broadly accelerated applications or individual algorithms to a new paradigm of smaller accelerator complexes [24, 51].

5.4 Datacenter Taxes

Figure 5 shows the percentage of CPU cycles per fine-grained category within the datacenter taxes. Within these taxes, protobuf, compression, and RPCs have widespread impact showing that *accelerating protobuf, compression, and RPC will achieve pareto benefits for the broader big data processing domain.* Next, we provide more details on each of these components.

Protobuf takes 20% to 25% of the datacenter tax suggesting that recently proposed protobuf accelerators such as [30, 42] could be beneficial in combination with software optimization approaches to reduce overhead. Spanner and BigTable have lower protobuf usage compared to BigQuery. This is due to the use of optimized file and data types that are flattened (versus typical protobuf structures) and compute reduction techniques like filter pushdowns.

We see large compression costs, ranging from 14% to 31%. In particular, compression takes more than 30% of datacenter tax in BigQuery and BigTable because both platforms operate on large chunks of compressed data, wherein compression and decompression are on the critical path. Thus compression accelerators will show strong benefits for these platforms that operate closely with the underlying data [6].

RPC costs are also high in database platforms, taking 23% and 37% in Spanner and BigTable, respectively. Since these platforms are often serving data to other platforms, RPCs are needed to feed the data obtained to other frontend services. In contrast, RPC overhead is relatively low at 11% in BigQuery because its queries are generally larger as compared to the database platforms. This result suggests that RPC acceleration is another candidate for acceleration gains.

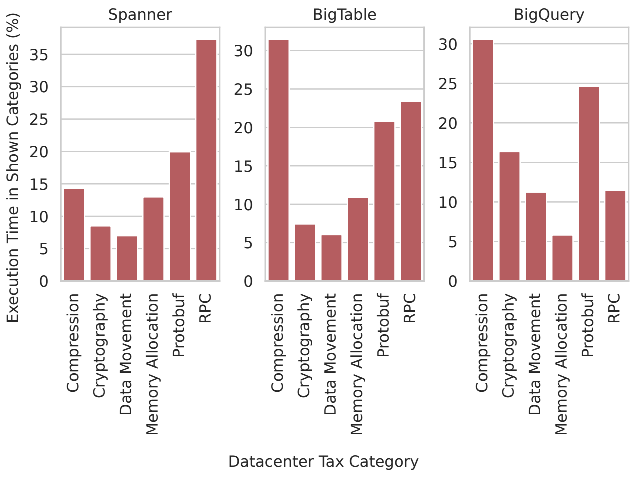


Figure 5: Datacenter Tax Execution Breakdown

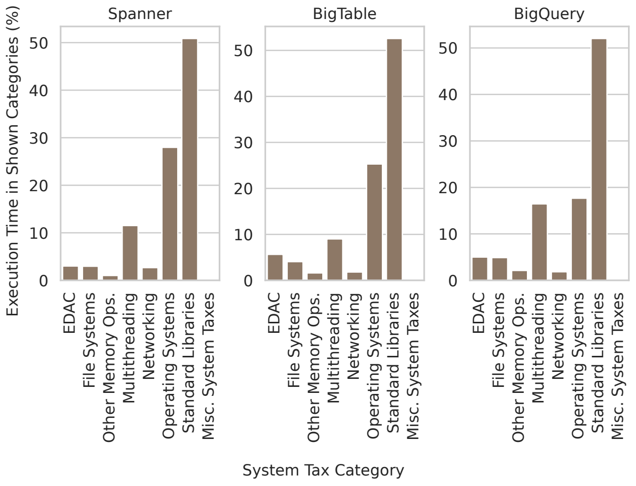


Figure 6: System Tax Execution Breakdown

5.5 System Taxes and Combined Acceleration

Figure 6 shows the percentage of compute cycles used for system taxes. Two fine-grained categories stand out: *operating systems and file systems.*

Across all platforms, we observe a high use of operating systems consuming 18% to 28% of system tax cycles. Standard libraries are also large for many platforms taking up to 53% of system tax. These overheads break down into many mixed functions that all Google platforms use extensively.

Given the high percentages of datacenter and system taxes, we conclude that accelerating data processing platforms depends on optimizing these components, along with the integration of core compute accelerators. One promising direction is to *build a set of "glue accelerators" that provides hardware acceleration for key datacenter taxes, such as protobuf and compression, in combination with common system taxes.* For example, these accelerators can fetch and prefetch data from distributed storage systems, apply datacenter tax acceleration, and store the resulting in-memory representation of data in a disaggregated cache tier. Furthermore, *integration with the core compute accelerators targeting scan, filter, and aggregation* can provide even larger benefits, such as operating on cached in-memory data. In a centralized accelerator-as-a-service model, this allows offloading core compute operators and "glue logic" to this complex for data processing platforms while allowing other non-data-processing services to re-use shared accelerators for better utilization. We model the benefit of the sea of accelerators complex in Section 6 under different acceleration execution models. However, next, we further breakdown node-level execution into the microarchitectural characteristics of these components.

Table 6: Platform IPC and MPKI Statistics

Statistic	Spanner	BigTable	BigQuery
IPC	0.7	0.7	1.2
Misses Per Kilo Instructions (MPKIs)			
BR	5.5	6.2	3.5
L1I	19.0	18.2	11.3
L2I	9.7	11.5	4.6
LLC	1.2	1.3	1.0
ITLB	0.5	0.5	0.4
DTLB LD	2.3	2.9	1.8

5.6 Microarchitectural Characterization

Tables 6 and 7 present microarchitecture performance data for the big data processing platforms. We present the following takeaways from these numbers.

First, the average instructions per cycle (IPC) of all big data processing platforms is 0.8. Spanner and BigTable exhibit IPCs lower than the average IPC, while BigQuery is higher. This indicates that *data analytics platforms are more conducive to run on accelerated machines with smaller, more energy-efficient architectures.*

Second, the two database platforms suffer from almost 2x higher branch, L1I, and L2I misses per kilo instructions (MPKI) as compared to the query engine. This suggests that *these database platforms have more complex control flows and larger instruction footprints than data analytics platforms.* This behavior is to be expected since databases typically have stricter performance and fault-tolerance SLOs, requiring the use of complex consensus and replication strategies resulting in long code paths and hard-to-predict control flows.

Third, the database platforms incur more DTLB Load MPKI than the query engine showing more back-end stalls while accessing data. This behavior is expected as data analytics jobs typically run large operations like scans and aggregations that have more uniform and predictable data access patterns while databases typically execute point queries and transactions, often with very little inter-query locality. These trends indicate that *heterogeneity can be beneficial for these workloads.* More complex cores with better branch predictors, larger instruction caches, better prefetchers, and larger TLB hierarchies are more suited to database workloads, while relatively simpler cores are more suited to running data analytics workloads [57].

Table 7 further breaks down the microarchitectural statistics into core compute (CC), datacenter taxes (DCT), and system taxes (ST). For BigQuery, core compute operations experience higher IPCs when compared to datacenter tax or system tax operations. The higher IPC results from both lower front-end stalls, as evidenced by fewer branch mispredictions, and instruction cache misses, and also from fewer back-end stalls, as evidenced by fewer DTLB misses.

These trends suggest that *code paths in core compute operations are shorter and less complex than the ones seen in tax operations and are more amenable to simpler cores.* Since tax operations handle complex tasks like network communication, compression, and encryption, it is expected that these code paths are more complex than core compute. This is positive news for future system designs: *when tax operations are offloaded to accelerators, the remaining core compute is amenable to traditional hardware optimizations.*

Table 7: High-Level Platform IPC and MPKI Statistics. CC, DCT, and ST, stand for Core Compute, Datacenter Tax, and System Tax, respectively.

	Spanner			BigTable			BigQuery		
	CC	DCT	ST	CC	DCT	ST	CC	DCT	ST
IPC	0.9	0.6	0.7	0.6	0.6	0.7	1.4	1.0	1.0
Misses Per Kilo Instructions (MPKIs)									
BR	5.4	5.5	5.5	5.2	5.3	6.9	2.0	3.8	3.5
L1I	12.4	16.7	21.6	9.6	14.7	21.9	1.1	13.6	10.8
L2I	4.2	8.0	11.8	4.2	8.4	14.7	0.4	3.4	6.0
LLC	0.6	1.0	1.4	1.0	1.2	1.4	0.3	1.1	1.1
ITLB	0.2	0.6	0.4	0.2	0.5	0.5	0.1	0.6	0.2
DTLB LD	0.8	2.0	2.7	1.3	2.1	3.6	0.6	2.2	1.7

6 SEA OF ACCELERATORS: LIMITS STUDY

This section presents analytical models for the sea of accelerators complex and a set of limit studies to estimate how accelerator system variations can improve end-to-end platform performance.

6.1 Base Model

With large hyperscale systems, it is useful to estimate performance gain attributed to new innovations in accelerator design, storage technologies, and networking capabilities before spending significant amount of engineering resources. We propose an analytical model that estimates the upper-bound performance benefit of acceleration for these platforms. It answers two system and architectural-level questions. First, how much can software-hardware co-design reduce distributed overheads? Second, how much benefit is achievable using a sea of accelerators complex for CPU execution?

Figure 7 shows the parameters for modeling execution time including overlaps and dependencies between accelerated and non-accelerated components. First, the model captures CPU time overlap with non-CPU dependencies such as IO and remote work as described in Section 4.1. Next, since accelerators can be invoked synchronously or asynchronously (sequential or parallel execution), the model includes overlap between each accelerated component. Asynchronous execution assumes that there is no dependency between dominant CPU components being accelerated and represents the ideal case where all accelerators are being executed in parallel, whereas synchronous execution represents a strict serial dependency between the core and other accelerators. Finally, the model incorporates on-chip and off-chip accelerator locations.

Equation 1 shows the definition of end-to-end time, t_{e2e}, as a function of CPU time, t_{cpu}, and its non-CPU dependencies (i.e., remote work or IO costs), t_{dep}. In this equation, $(1-f)*min(t_{cpu}, t_{dep})$ accounts for the overlapped time between CPU and non-CPU dependencies and is subtracted to achieve the end-to-end time. Equation 2 is an extension of Equation 1 used to calculate a new accelerated end-to-end time, t'_{e2e}, as a function of accelerated CPU time, t'_{cpu}. This accelerated CPU time is split into accelerated and non-accelerated

Time Parameters	
t_{e2e}, t'_{e2e}	Original and accelerated end-to-end time (s)
t_{cpu}, t'_{cpu}	Original and accelerated CPU time (s)
t_{dep}	Non-CPU time (s) that t_{cpu} depends on
t_{acc}	Accel. CPU time (s) for all subcomponents
t_{nacc}	Unaccel. CPU time (s) for all subcomponents
t_{sub_i}, t'_{sub_i}	Original and accel. CPU subcomp. time (s)
t_{lsub}	Largest accelerated CPU subcomp. time (s)
t_{pen_i}	Accelerator penalty time (s)
t_{setup_i}	Setup time (s) for the accel. (e.g., initialization)

Overlap Parameters	
f	Sync. factor between t_{dep} and t_{cpu} from $[0, 1]$
g_{sub_i}	Sync. factor between t'_{sub_i}'s from $[0, 1]$

Miscellaneous Parameters	
N, U	Number of non-accel. and accel. components
s_{sub_i}	Acceleration factor for a CPU subcomponent
B_i	Bytes to offload to accelerator (0 when on-chip)
BW_i	Bandwidth between CPU and accelerator

$$t_{e2e} = t_{cpu} + t_{dep} - (1 - f) * min(t_{cpu}, t_{dep}) \quad (1)$$

$$t'_{e2e} = t'_{cpu} + t_{dep} - (1 - f) * min(t'_{cpu}, t_{dep}) \quad (2)$$

$$t'_{cpu} = t_{acc} + t_{nacc} \quad (3)$$

$$t_{nacc} = \sum_{i=0}^{N} t_{sub_i} \quad (4)$$

$$t_{acc} = max((\sum_{i=0}^{U} g_{sub_i} * t'_{sub_i}), t_{lsub}) \quad (5)$$

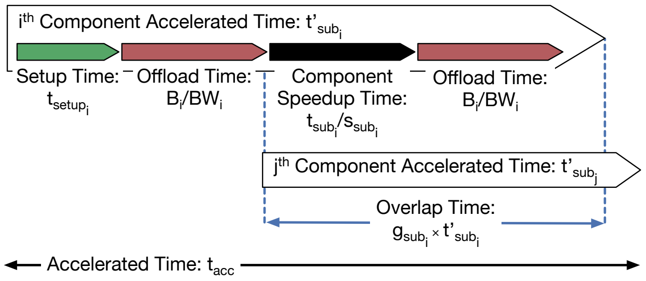
$$t_{lsub} = max(\{t'_{sub_i} : i = 0, ..., U\}) \quad (6)$$

$$t'_{sub_i} = \frac{t_{sub_i}}{s_{sub_i}} + t_{pen_i} \quad (7)$$

$$t_{pen_i} = t_{setup_i} + 2 * \frac{B_i}{BW_i} \quad (8)$$

Figure 7: Base Model Parameters and Equations

time, t_{acc} and t_{nacc}, respectively, as shown in Equation 3. Here the non-accelerated time, t_{nacc}, is a sum of all N original unaccelerated component times, t_{sub_i} (i.e., unaccelerated time to complete compression or aggregation compute). Equation 5 shows the accelerated CPU time, t_{acc}, as function of accelerated subcomponent time, t'_{sub_i}, and a corresponding overlap factor, g_{sub_i}, for all accelerated components U. Here the g_{sub_i} overlap factor indicates the overlap of an accelerated component with all other execution components. When all accelerated components overlap, then the largest accelerated subcomponent, t_{lsub}, dominates, as seen in Equation 6. The accelerated subcomponent time, in this case, is the original CPU component time, t_{sub_i}, sped up by s_{sub_i} and delayed by a penalty time of t_{pen_i} shown in Equation 7. The penalty time represents accelerator setup time t_{setup} (e.g., initializing accelerator-specific



ith Component Accelerated Time: t'sub_i
Setup Time: tsetup_i
Offload Time: B_i/BW_i
Component Speedup Time: tsub_i/ssub_i
Offload Time: B_i/BW_i
jth Component Accelerated Time: t'sub_j
Overlap Time: gsub_i × t'sub_i
Accelerated Time: tacc

Figure 8: t_{acc} Diagram. Example of Eq. 5-8 with two components (t_{sub_i} and t_{sub_j}) accelerated.

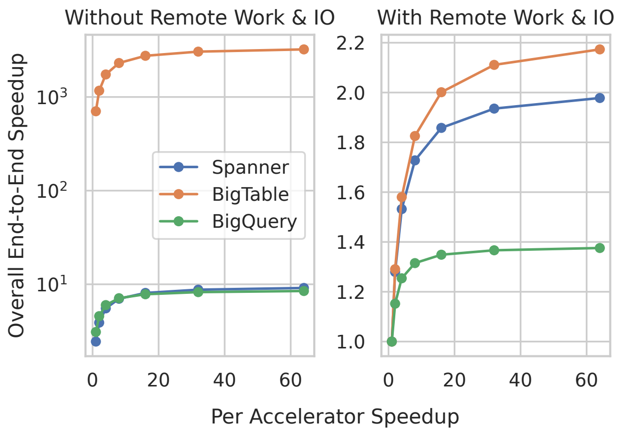


Figure 9: Synchronous On-Chip Upper Bound. 0% or 100% of remote work and IO (i.e., non-CPU) time is removed.

registers) combined with the data transfer time of communicating from the host memory space to the accelerator. In the case of an on-chip shared-memory-coherent accelerator, all of its data is already present in the cache and/or DRAM, so the penalty time would only be t_{setup} (i.e., B_i is 0). Off-chip uncached accelerators on the other hand would need to transfer B_i bytes of data over an off-chip link that has a BW_i bandwidth. Figure 8 pictorially shows the accelerated CPU time, t_{acc}, as a function of any penalties, overlaps, and speedups in Equations 5-8 with two components accelerated.

6.2 On-Chip Acceleration Limit Studies

In this set of studies, we measure the upper-bound performance speedup when accelerating the dominant CPU components identified in Section 5, through *on-chip* acceleration. To consider the impact of potential non-CPU optimizations (i.e., retrieving data over the network or remote shuffle costs), we keep or remove non-CPU time (t_{dep}) from the system. For the components to accelerate, we chose the top datacenter taxes (compression, RPC, protobuf), system taxes (STL, OS), and core compute for each platform (read, filter, compute, compaction, write, aggregation, misc. core operations). For experiment simplicity, we assume that all CPU components are accelerated from 1x to 64x (s_{sub_i}) in lockstep, everything is on-chip (off-chip bytes transferred B_i is 0), and the accelerator setup penalty

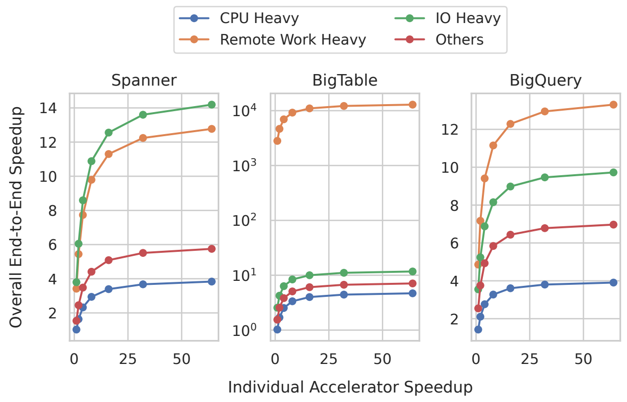


Figure 10: Grouped Synchronous On-Chip Upper Bounds. Remote work and IO time are all removed.

New Time Parameters	
t_{chnd}	Accelerated chained component time (s)
t_{lpen}	Largest accelerator penalty time (s)
t_{lsubnp}	Largest accelerator component time without penalty time (s)

New Miscellaneous Parameters	
C	Number of chained compute components

$$t'_{cpu} = t_{chnd} + t_{acc} + t_{nacc} \quad (9)$$

$$t_{chnd} = t_{lpen} + t_{lsubnp} \quad (10)$$

$$t_{lpen} = max(\{t_{pen_i} : i = 0, ..., C\}) \quad (11)$$

$$t_{lsubnp} = max(\{\frac{t_{sub_i}}{s_{sub_i}} : i = 0, ..., C\}) \quad (12)$$

Figure 11: Model Extension For Accelerator Chaining

(t_{setup}) is 0. We also assume that all components are synchronous with respect to one another (g_{sub_i} is 1) representing that accelerator execution cannot be parallelized with other execution since shared memory synchronization is traditionally costly. Finally, the values of f, t_{e2e}, t_{sub_i}, and t_{dep} are derived from Sections 4 and 5.

Figure 9 shows the upper bound speedup of end-to-end execution time for each platform's queries with and without non-CPU dependencies. With the removal of non-CPU dependencies, the ideal upper bound speedup for all queries can reach peaks of 9.1x, 3,223.6x, and 8.5x for Spanner, BigTable, and BigQuery, respectively. However, if the dependencies still exist, we see multiple orders-of-magnitude lower theoretical upper bounds of 2.0x, 2.2x, and 1.4x, for Spanner, BigTable, and BigQuery, respectively. This result clearly demonstrates that hardware-only acceleration can only achieve a fraction of the upper-bound performance in these distributed platforms. A software-hardware co-design shifts the IO/remote bottleneck to the CPU and, therefore, drastically improves the speedup obtained.

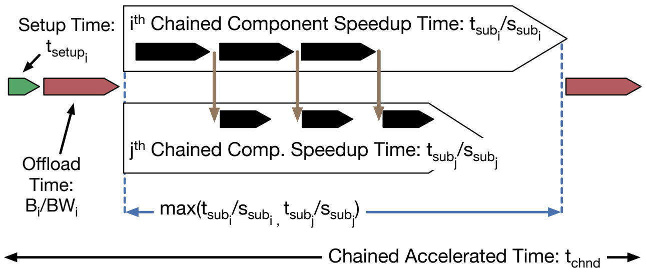

Setup Time: t_{setup_i}
i^{th} Chained Component Speedup Time: t_{sub_i}/s_{sub_i}
Offload Time: B_i/BW_i
j^{th} Chained Comp. Speedup Time: t_{sub_j}/s_{sub_j}
$max(t_{sub_i}/s_{sub_i}, t_{sub_j}/s_{sub_j})$
Chained Accelerated Time: t_{chnd}

Figure 12: t_{chnd} Diagram. Example of Eq. 10 with two components (t_{sub_i} and t_{sub_j}) chained and $t_{pen_i} > t_{pen_j}$.

Figure 10 further breaks down the speedup without non-CPU dependencies into the four query groups given in Section 4: "CPU Heavy", "Remote Work Heavy", "IO Heavy", and "Other" queries. Due to IO and remote work removal, we see that query groups that are IO or remote heavy dominant have the largest speedups across all platforms. Removing these dependencies increases the initial speedup when acceleration is close to 1x, while the remaining CPU time that is accelerated affects the overall slope of acceleration. For Spanner and BigTable, removing IO and remote work is important for large speedups across all queries. However, unlike the databases, BigQuery's execution time is more varied, thus removal of IO and remote work, as well as CPU acceleration, are all equally necessary for substantial performance gains.

6.3 Accelerator System Features Limit Study

We next study the sea of accelerators with different execution models. For this set of analyses, we vary accelerator placement – on-chip or off-chip – and accelerator invocation by the core – synchronous or asynchronous. Additionally, we evaluate a "chained" execution model where the accelerators can directly communicate with each other. This model provides an unique opportunity to pipeline accelerators: while the current accelerator is still processing, the computed results are sent to the next accelerator, allowing it to overlap execution with prior accelerators while still maintaining the strict dependency between components. We first extend our analytical model to represent accelerator chaining and then analyze the benefits of chaining when compared to more traditional accelerator models. We then conclude with a setup time limit study and prior accelerator comparison under the different system features.

6.3.1 Extending to Chained Acceleration. For operations that are known to be linked together, accelerator chaining can improve the performance benefits by avoiding communication latency to the CPU when sending data between accelerators. Figures 11 and 12 show the model extension to account for a subset of accelerated components being chained. Equation 9 shows the modification of overall new CPU time, t'_{cpu}, to include the time spent in unchained accelerated components (the original accelerated time t_{acc}), unaccelerated components (t_{nacc}), and t_{chnd}, a new variable to model chained accelerated component time. For the chained acceleration time, all C chained accelerators will be pipelined, with the longest accelerated component without accelerator setup penalty, t_{lsubnp}, determining the overall time of the chain as seen in Equation 12.

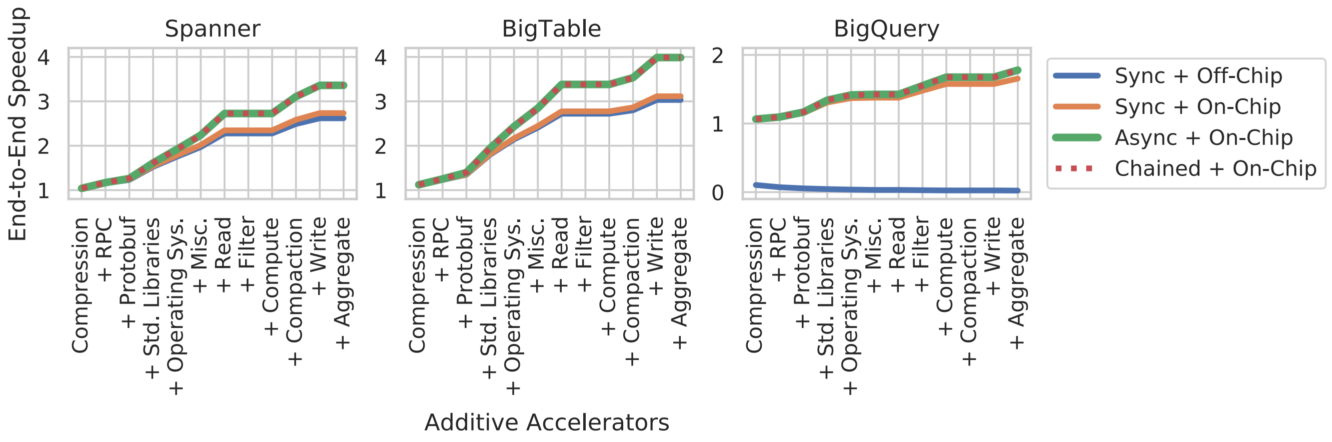


Figure 13: Accelerator Feature Upper Bounds. Subsequent X-axis elements are incrementally accelerated. Remote work and IO are *not* removed.

Finally, we bound the initial penalty time for setting up the accelerator chain by the largest accelerator penalty time, t_{lpen}, and add it to longest accelerated component (t_{lsubnp}) to obtain the overall chained execution time (t_{chnd}).

6.3.2 Accelerator Feature Upper Bounds. Figure 13 evaluates four accelerator configurations, starting with the traditional synchronous off-chip accelerators and then incrementally adding optimizations. First, we remove off-chip data movement by moving the accelerator on-chip (`Sync + On-Chip`). Then, we assume that the accelerator is asynchronous to other accelerators improving concurrency (`Async + On-Chip`). Finally, we evaluate the impact of accelerators chained together (`Chained + On-Chip`): this removes communicating back to CPU after every execution and avoids the need for fine-grained synchronization via shared memory. For changing synchronicity, we vary g_{sub_i}, which is 0 for synchronous case and 1 for asynchronous case. For accelerators located off-chip, we calculate off-chip data transfer overheads by setting each CPU component's B_i to the average number of bytes in a query, then divide by a PCIe Gen5 link bandwidth BW_i of $4GB/s$. For this experiment, we progressively add the set of accelerators mentioned in Section 6.2, beginning with datacenter tax operations, then system tax, and core compute operations shown in subsequent X-axis labels.

Starting with Spanner and BigTable, we see that synchronous on-chip acceleration provides a 1.04x performance uplift over off-chip acceleration. Moving on-chip has low benefit since the majority of queries transfer a small amount of data. However, in scenarios where an off-chip accelerator provides a larger speedup factor, this benefit would be larger as on-chip acceleration would amortize more of the off-chip data transfer. With asynchronicity among accelerators (`Async + On-Chip`), the end-to-end speedup improves up to 1.3x compared to synchronous execution indicating that asynchronicity is critical in multi-accelerator systems.

When full asynchronicity is bounded by the inherent serialization between components, chaining provides an alternative if accelerators can send data to one another. For Spanner and BigTable, *chaining provides less than 1% difference to fully asynchronous accelerators.* Chaining amortizes pipeline penalty time (t_{pen_i}) and allows for fast data transfer between accelerators through constructs like pipeline FIFOs instead of enforcing complex fine-grained synchronization through shared memory.

BigQuery shows different trade-offs due to its larger data payloads. BigQuery, as a data analytics platform, often operates on orders of magnitude larger batches of data per query compared to transactional platforms like Spanner and BigTable. Thus, the data transfer off-chip dominates, leading to a 0.02x slowdown for off-chip acceleration. Moving acceleration on-chip is beneficial since the penalty is removed. Once on-chip, BigQuery shows similar trade-offs with asynchronous and chained accelerators, with speedups reaching 1.8x over a non-accelerated baseline.

Overall, we find that *off-chip acceleration is beneficial when it significantly accelerates a large portion of program execution to amortize any offload penalties.* Furthermore, *while full asynchronicity among accelerators is untenable, accelerator chaining is a practical way to realize much of the asynchronous performance benefits.*

6.3.3 Setup Time Analysis. In this study, we measure the impact of accelerator setup time on end-to-end speedup. Figure 14 shows the effect of setup time across the platforms under the accelerator configurations mentioned in Section 6.3.2. We vary the setup time of the accelerators mentioned in Section 6.2, with an 8x speedup (s_{sub_i}) per accelerator. For Spanner and BigTable, increasing setup time in the synchronous setups can lead to large slowdowns due to the setup penalties applied to each accelerator invocation. Once moving to an ideal asynchronous execution upper bound we see a large improvement since the setup time is parallelized with accelerator invocation. For BigQuery, we see that off-chip penalties to copy the data dominate due to its larger working sets. Once acceleration is on-chip, we see similar performance degradation when the setup time is large enough.

6.3.4 Prior Accelerator Comparison. In this study, we show the speedup gained with a subset of published on-chip accelerators

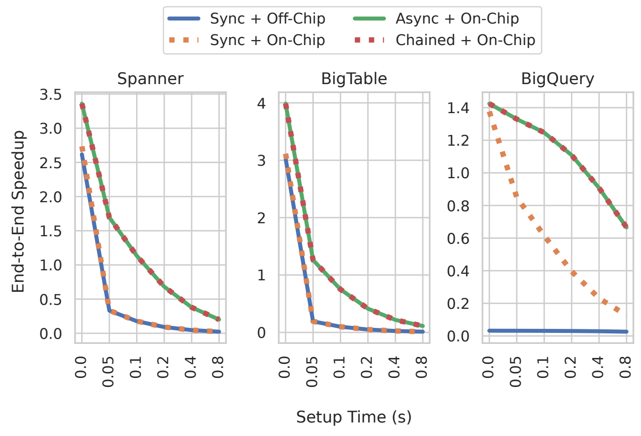


Figure 14: Setup Time Sweep

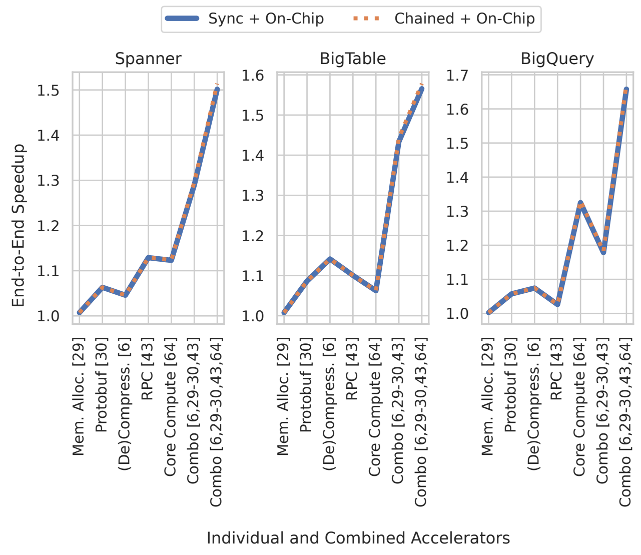


Figure 15: Prior Accelerator Comparison

using the proposed model in a synchronous and chained setup (`Sync/Chained + On-Chip`). We used the accelerators with the largest published speedup for their respective operations for all core compute operations [64] and the following datacenter taxes: memory allocation [29], protobuf serialization/deserialization [30], RPC [43], and compression/decompression [6]. For all accelerators, we obtain the speedup (t_{sub_i}) published and maintain uniformity by zeroing the setup time (t_{setup_i} is 0) since the metric was not universally reported. We see in Figure 15 that holistic synchronous acceleration can yield a 1.5x-1.7x speedup across the data processing platforms. When expanding to chained acceleration, we see limited benefit since the sped up memory allocation component serves as the critical bottleneck of the pipeline. However, as shown in Section 6.3.2, there is a high potential for speedup if we can achieve fully pipelined and balanced execution across the accelerator chain.

Table 8: Model Validation Results

Measured RISC-V RTL Results	
Proto. Ser. t_{sub_i}, s_{sub_i}, t_{setup_i}	518.3μs, 31x, 1,488.9μs
SHA3 t_{sub_i}, s_{sub_i}, t_{setup_i}	1,112.5μs, 51.3x, 4.1μs
Non-Accel. CPU t_{sub_i}	4,948.7μs
Proto. Ser. B_i	0
Proto. Ser. t_{dep}	0
SHA3 B_i	0
SHA3 t_{dep}	0
Measured chained execution t'_{e2e}	6,075.7μs

Model Estimated Results	
Modeled chained execution t'_{e2e}	6,459.3μs

6.4 Model Validation and Limitations

We evaluate the effectiveness of the chained model using a heterogeneous accelerator RISC-V system-on-chip built with the Chipyard RTL framework [8] and simulated using the FireSim FPGA-accelerated RTL simulator [31]. Our evaluation contains an open-source protobuf serialization [30] and SHA3 accelerator [50] connected to separate in-order Rocket cores and a third Rocket core without accelerators. To calculate model parameters, we built a synthetic experiment using three Linux benchmarks built off of HyperProtoBench [30], where we first serialized identical fleet-wide representative protobuf messages then computed their SHA3 hash. In all cases, the protobuf messages and intermediate results fit on-chip and require no IO (t_{dep} and B_i are 0). We measure the t_{sub_i} subcomponent times of serialization, hashing, and non-accelerated time, using a non-accelerated synchronous benchmark where all protobufs are serialized before hashing. In the next benchmark, we accelerate serialization and hashing and measure the speedup (s_{sub_i}) and setup penalties (t_{setup_i}) of each accelerator. The final benchmark then emulates chained acceleration with each accelerator operating on a single element on parallel Linux threads and is used to give an experimental end-to-end time (t'_{e2e}) number to compare against the model.

Table 8 shows the model parameters estimated and measured. We see that SHA3 compute takes longer than the serialization compute when both run on CPU at 1,112.5μs and 518.3μs, respectively. Additionally, the overall CPU subcomponent time, t_{sub_i}, is over 4x larger than either component due to initializing protobuf messages, Linux threading and multiprocessing overheads, and measurement overheads. Once accelerated, we see a speedup of 31x and 51.3x for protobuf serialization and SHA3 hashing, respectively, with a lower setup time for SHA3 hashing. This is due to protobuf serialization needing to allocate a memory area for serialized messages. Using these numbers we can then use Equations 9 and 10 to estimate a chained execution time of 6,459.3μs. When compared to the measured 6,075.7μs, we see that the model is within a 6.1% difference, validating the model in this one case.

While the synthetic validation setup serves as an initial scoped implementation of software-centric accelerator chaining, future work is needed to validate the model with additional synthetic data, different accelerator placements, and an implementation of hardware-centric accelerator chaining. Furthermore, while our validation assumes serialization and hashing are candidates for chaining, not all production code will have sequential code patterns, thus careful identification of common sequential patterns and/or code modifications is required for chaining viability. For simplicity, our limit studies also only focus on fully synchronous or asynchronous accelerators and non-CPU dependencies but can be later expanded to cover various amounts of synchronization between CPU components and non-CPU dependencies. Finally, while some of our limit studies assume equivalent acceleration speedups per accelerator, different components can have varied speedups leading to more nuanced improvements for the platforms. Even with these limitations, we believe that our validated model allows us to do complete design space explorations of different acceleration strategies using detailed production traces.

7 RELATED WORK

To the best of our knowledge, our work presents the first holistic study of big data processing platforms at a large datacenter, including system balance, execution breakdown, and limit analysis with a proposed chained acceleration model.

Prior Profiling Studies: Previous works [40, 54, 55] breakdown analytical and transactional databases, identifying high instruction footprints and frontend stalls, but were limited to standard benchmark suites. Our work differs by focusing on live production traffic on large-scale distributed data processing, including the relevance of datacenter taxes and high scan-aggregation versus join costs on these workloads [17]. CloudSuite and subsequent studies [19, 66] focus on scale-out cloud service workloads, with a focus on an open-source NoSQL database. Our work instead looks at production SQL, NoSQL, and query engine platforms, showing how CPU cycle utilization and microarchitectural characteristics differ between them within both the core compute and overheads. Additionally, we profile production workloads running live traffic instead of open-source benchmarks due to differences in workload structure and size. Prior work done in [44] focuses on processor performance improvement of older commercial workloads running on the Oracle commercial database unlike hyperscale distributed databases in this work. Hyperscale data center providers have done fleet-wide analysis of workloads [28, 56], but our work takes an alternative vertical profiling approach of data processing platforms to show the impact of core data compute and increases in datacenter taxes within these platforms as compared to the entire fleet.

Hyperscale Hardware Accelerators: Research studies on accelerator design for data center applications are prominent. Prior studies [27, 45] focus on domains like machine learning and video processing, unlike data processing in this work. Acceleration hardware such as [1, 6, 26, 29, 30, 42, 43] target datacenter taxes within hyperscalers that complement our work. However, we holistically emphasize the combination of these hardware targets with further core compute acceleration. Studies [5, 7, 18, 21, 32, 62–64] accelerate database-specific kernels such as scan, aggregation, joining, and filtering. These are relevant, but as our study points out, they need to be taken into account in the context of larger holistic distributed systems and system balance trends. There has been very little emphasis on optimizing query engines in a storage disaggregated system, with some recent work looking at system issues primarily [65].

Sea Of Accelerator Chaining: Dataflow architectures [22, 64], CGRAs [16, 49], VLIW architectures [46, 47], systolic array architectures [33], and vector machines [10, 48] also share similarities to accelerator chaining. We differ in that our compute units are larger heterogeneous accelerators chained together, and all acceleration is completed and managed by hardware instead of a compiler. This implies that the programmers would need to pass both the operation type and their dependence information to invoke the accelerator chain.

8 CONCLUSION

In this paper, we identify systems and hardware acceleration opportunities in Google's distributed databases and analytics engines by characterizing the major bottlenecks in their execution time. We find that remote work and IO dominate over 52% of the end-to-end execution time, as horizontal scaling to millions of servers depends on distributed storage and inter-node communication. Therefore, hardware-software co-design, which optimizes IO and remote work in addition to compute acceleration, is critical for these platforms. While our profiling shows that no single core compute function accounts for most of the execution time, a "sea of accelerators" collectively can accelerate groups of key data processing and tax functions. Our analytical modeling demonstrates the potential gain of a data processing sea of accelerators and analyzes the trade-offs between various accelerator execution models. Modeling results show that removing the CPU invocation overhead by chaining accelerators can lead to over a 3x speedup in these data processing platforms over the baseline.

We make a case for a sea of accelerators complex for hyperscale data processing and hope the community will explore the architectural and software-hardware co-design space for such an accelerator complex for future at scale systems.

ACKNOWLEDGMENTS

This work builds on prior profiling work done by several engineering teams and interns at Google (e.g., GWP, Spanner, BigTable, BigQuery), and we would like to thank people within those teams, including Arif Arman, Tae Jun Ham, and Yixin Luo. We would also like to thank Andy Caldwell, Liqun Cheng, Urs Holzle, Sam McVeety, Tipp Moseley, Fatma Ozcan, Jeff Shute, Daniel Stodolsky, Chris Taylor, Amin Vahdat and the anonymous reviewers and artifact evaluators for their paper feedback. We would also like to thank Tianrui Wei for validation feedback.

This research was supported by the SLICE Lab industrial sponsors and affiliates and by the NSF CCRI ENS Chipyard Award #201662. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix describes how to reproduce the model validation results in Section 6.4. First, we will use a FireSim FPGA-accelerated simulation of a Chipyard-based RISC-V system-on-chip (SoC) to cycle-exactly simulate software chaining of protobuf serialization with SHA3 hashing. This involves first booting Linux on this system, then running protobuf serialization on a selected batch of protobuf messages, followed by chaining the serialization output with SHA3 hashing to collect performance metrics. Afterward, we pass the obtained metrics into the provided Python implementation of the analytical performance model given in Figures 7 and 11 to obtain the estimated end-to-end execution time.

A.2 Artifact Checklist

- **OS Environment:** AWS FPGA Developer AMI 1.6.1.
- **Hardware:** AWS EC2 instances: 1x `c5.9xlarge` and 3x `f1.2xlarge` instances.
- **Disk Space Needed:** 300GB (on EC2 instances).
- **Experiments:** Replicate Table 8.
- **Setup time:** 1.5 hours (scripted installation).
- **Experiment time:** 1 hour (scripted run).
- **Publicly available:** Yes.
- **Licenses:** Multiple, see downloads in Appendix Section A.3.
- **Archived:** See downloads in Appendix Section A.3.

A.3 Descriptions

The artifact consists of nine `Git` repositories stored in `Zenodo` archives:

(1) **firesim-protoacc-sha3-ae**: Top-level FireSim simulation environment.
(https://doi.org/10.5281/zenodo.7814284)
(2) **chipyard-protoacc-sha3-ae**: Chipyard RISC-V SoC generation environment.
(https://doi.org/10.5281/zenodo.7814222)
(3) **rocket-chip-protoacc-sha3-ae**: Rocket Chip RISC-V generation library.
(https://doi.org/10.5281/zenodo.7814238)
(4) **riscv-torture-protoacc-sha3-ae**: Patched RISC-V torture tests.
(https://doi.org/10.5281/zenodo.7814265)
(5) **protoacc-protoacc-sha3-ae**: Protobuf accelerator design, scripts, and software.
(https://doi.org/10.5281/zenodo.7814245)
(6) **protoacc-sha3-sw**: Protobuf and SHA3 accelerator chained and unchained software used for measurements.
(https://doi.org/10.5281/zenodo.7814225)
(7) **firemarshal-protoacc-sha3-ae**: Linux build scripts.
(https://doi.org/10.5281/zenodo.7814260)
(8) **riscv-linux-protoacc-sha3-ae**: Patched Linux.
(https://doi.org/10.5281/zenodo.7814266)
(9) **profiling-data-processing-model-isca23-ae**: Python implementation of analytical model.
(https://doi.org/10.5281/zenodo.7814235)

Users need not download the bottom eight repositories manually since they will be obtained automatically in the download setup scripts.

A.4 Hardware and Software Dependencies

One AWS EC2 `c5.9xlarge` instance (also referred to as the "manager" instance), and three `f1.2xlarge` instances are required. The `f1.2xlarge` instances will be launched automatically by the FireSim manager instance. All machines will be configured to use 300GB of disk space. To optionally run FPGA builds (see Appendix Section A.8), you will need one `z1d.6xlarge` instance. However, we provide a pre-built FPGA image to avoid the long latency (10 hours) of building a fresh FPGA image. No software dependencies are required other than an `ssh` client. All other requirements are installed by the setup scripts.

A.5 Installation

First, follow along with the instructions on the FireSim website[2] to create a manager instance on AWS EC2. You must complete up to and including "Section 2.3.1.2: Key Setup, Part 2", with the following changes in "Section 2.3.1":

(1) When instructed to launch a `c5.4xlarge` instance, choose a `c5.9xlarge` instead.
(2) When entering the `root` EBS volume size, use 300GB.
(3) Do not paste any information into the "Advanced Details" text box.

Once you have completed up to and including "Section 2.3.1.2" in the FireSim documentation, you should have a manager instance set up, with an IP address and key. Use `ssh` (or optionally `mosh`) to login to the instance.

From this point forward, all commands should be run on the manager instance that you `ssh`'ed into. Next, download the top-level FireSim simulation repository, like so:

```
$ cd ~
# Enter as a single line
$ wget -O firesim-protoacc-sha3-ae.zip https://zenodo.org/
    ↪ record/7814284/files/firesim-protoacc-sha3-ae.zip
$ unzip firesim-protoacc-sha3-ae.zip
$ cd firesim-protoacc-sha3-ae
```

Next, run the following, which will initialize the machine and all software requirements (i.e., downloading software packages or installing FPGA runtime requirements). It is recommended that you run the command within `screen` or `tmux` so that any disconnections to the manager instance do not cancel the setup:

```
$ cd scripts
$ sudo ./machine-launch-script.sh
```

To ensure that this step completed successfully, you can verify that the `machine launch script completed` output is present in the `/home/centos/machine-launchstatus` file. Next, make sure to completely close all `ssh/screen/tmux` sessions, terminals, etc. to the machine and re-enter the machine. Next, run the following, which will initialize more dependencies and run basic FireSim and Chipyard setup steps (i.e., RISC-V and host toolchain installation).

[2] FireSim 1.12.0 documentation: https://docs.fires.im/en/1.12.0/

Similar to before, it is also recommended to run this step within a `ssh` or `tmux` session:

```
$ cd firesim-protoacc-sha3-ae
$ ./scripts/first-clone-setup-fast.sh
```

This step should take around 1.5 hours. Upon successful completion, it will print:

```
first-clone-setup-fast.sh complete
```

Once this is complete, run:

```
$ source sourceme-f1-manager.sh
```

Sourcing this file will have set up your environment to run Linux and other SoC simulations. Finally, finish setting up your manager by running:

```
$ firesim managerinit
```

Once completed, your manager instance is fully set up to run protobuf and SHA3 accelerator simulations.

A.6 Experiment Workflow

Now that the environment is setup, we will run the full artifact evaluation script, which does the following:

(1) On the manager instance, build the modified protobuf library and generate protobuf collateral needed for chained simulation.
(2) Build three Buildroot-based Linux distributions containing the protobuf collateral and SHA3 code needed to obtain results. This will be booted on the accelerated system.
(3) Run the three FireSim simulations, which do the following (each per simulation):
   (a) Launch an `f1.2xlarge` instance.
   (b) Copy simulation infrastructure to the F1 instance.
   (c) Run the benchmark provided (Linux distribution created).
   (d) Copy back the results to the manager instance.
   (e) Terminate the `f1.2xlarge` instance.
(4) Parse the output results and regenerate Table 8.

Now let's run the aforementioned full artifact evaluation script (again, it is recommended to run this command within a `screen` or `tmux` session):

```
$ ./full-ae.sh
```

This should take around 1 hour. When it completes, it prints:

```
Success
```

The FireSim manager will have automatically terminated any instances it launched during the process, but please confirm in your AWS EC2 management console that no instances remain beside the manager.

A.7 Evaluation and Expected Results

Next, let's view the output results generated from `full-ae.sh` in the previous section. Once you are finished running `full-ae.sh` you should be able to see the results printed to the terminal as well as in the file `final-ae-results.txt`. You can print the full results by running the following (or opening the file in a text editor):

```
$ cat final-ae-results.txt
```

You should see each of the measured values in Table 8 except for B_i and t_{dep}, which are assumed to be 0. Note that the times measured are in nanoseconds instead of microseconds and that the numbers are slightly different than paper numbers (within a 1% difference). This is due to a small amount of non-determinism introduced by Linux's measurement of time in the simulation.

Once your evaluation is complete, manually terminate your manager instance in the EC2 management console and confirm that no other instances from the evaluation process are left running.

A.8 Customization

Since the SoC is fully open-sourced and available online, users can change the configuration of the accelerators, run further experiments, or more. Please refer to the FireSim[3] and Chipyard[4] for more information on how to customize the design. Tutorial slides are also present through the FireSim website[5] for more modern versions of the tools. The core software chaining tests that are run are present in `$SW_DIR` and can be customized/improved as necessary. Both the protobuf and SHA3 accelerators are found in `$GEN_DIR/protoacc` and `$GEN_DIR/sha3`, respectively.

If modifications are made to the RTL, users need to re-build FPGA images. We provide a pre-built FPGA image for the design in this paper (generated from the included RTL), encoded in the configuration files in the artifact. Re-generating the supplied FPGA images can also be done by modifying the S3 bucket name in `$CFG_DIR/config_build.ini` to an unused bucket name (that the manager will create), then running `./buildafi.sh`. This will take around 10 hours, require one `z1d.6xlarge` instance, generate one new AGFI (i.e., a FPGA bitstream on EC2 F1), and place its `config_hwdb.ini` entry in `$BLT_DIR/<config-name>`. To use the new AGFI that was generated, replace the existing entry in the `$CFG_DIR/config_hwdb.ini` file (or, for a new config, add it). When an FPGA build completes, the FireSim manager will automatically terminate the instances it launched during the build process, but please confirm in your AWS EC2 management console that no instances remain beside the manager. More details about the FireSim FPGA build process can be found in the FireSim documentation. Note that many of the FireSim manager build configuration files are in a non-standard location to simplify scripting for artifact evaluation. Open `./buildafi.sh` to see their locations.

A.9 Methodology

Submission, reviewing, and badging methdology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://github.com/mlcommons/ck/blob/master/docs/artifact-evaluation/submission.md
- https://github.com/mlcommons/ck/blob/master/docs/artifact-evaluation/reviewing.md

[3]FireSim 1.12.0 Documentation: https://docs.fires.im/en/1.12.0/
[4]Chipyard 1.5.0 Documentation: https://chipyard.readthedocs.io/en/1.5.0/
[5]FireSim HPCA 2023 Tutorial: https://fires.im/hpca-2023-tutorial/

REFERENCES

[1] [n. d.]. Intel® QuickAssist Technology - NGINX* Performance White Paper. https://www.intel.com/content/www/us/en/content-details/767645/intel-quickassist-technology-nginx-performance-white-paper.html

[2] 2019. MariaDB foundation. https://mariadb.org/

[3] 2020. Rethink Data Report. https://www.seagate.com/files/www-content/our-story/rethink-data/files/Rethink_Data_Report_2020.pdf

[4] 2022. Data Lake Analytics. https://azure.microsoft.com/en-us/services/data-lake-analytics/#overview

[5] 2022. Working with AQUA (Advanced Query Accelerator). https://docs.aws.amazon.com/redshift/latest/mgmt/managing-cluster-aqua.html

[6] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA45697.2020.00012

[7] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, et al. 2017. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 245–258.

[8] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[9] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[10] Melvin C. August, Gerald M. Brost, Christopher C. Hsiung, and Alan J. Schiffleger. 1989. Cray X-MP: The birth of a supercomputer. *Computer* 22, 1 (1989), 45–52.

[11] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 331–343. https://doi.org/10.1145/3035918.3056103

[12] Josiah Carlson. 2013. *Redis in action*. Simon and Schuster.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[14] Google Cloud. 2012. An inside look at Google BigQuery. https://cloud.google.com/files/BigQueryTechnicalWP.pdf

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[16] Vidushi Dadu and Tony Nowatzki. 2022. TaskStream: accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1–13.

[17] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1206–1220.

[18] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. 2017. The Mondrian Data Engine. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 639–651.

[19] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[20] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[21] Sebastian Haas, Oliver Arnold, Stefan Scholze, Sebastian Höppner, Georg Ellguth, Andreas Dixius, Annett Ungethüm, Eric Mier, Benedikt Nöthen, Emil Matúš, et al. 2016. A database accelerator for energy-efficient query processing and optimization. In *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*. IEEE, 1–5.

[22] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Jae W Lee, David Bruns-Smith, Brendan Sweeney, Krste Asanovic, Young H Oh, and Lisa Wu Wills. 2021. Accelerating Genomic Data Analytics With Composable Hardware Acceleration Framework. *IEEE Micro* 41, 3 (2021), 42–49.

[23] Herodotos Herodotou and Elena Kakoulli. 2019. Automating Distributed Tiered Storage Management in Cluster Computing. *Proc. VLDB Endow.* 13, 1 (sep 2019), 43–56. https://doi.org/10.14778/3357377.3357381

[24] Mark D Hill and Vijay Janapa Reddi. 2021. Accelerator-level parallelism. *Commun. ACM* 64, 12 (2021), 36–38.

[25] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.

[26] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W Lee. 2020. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 322–334.

[27] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. https://arxiv.org/pdf/1704.04760.pdf

[28] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[29] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. *ACM SIGPLAN Notices* 52, 4 (2017), 33–45.

[30] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 462–478.

[31] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[32] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-Memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 468–479.

[33] Hsiang-Tsung Kung. 1982. Why systolic architectures? *Computer* 15, 01 (1982), 37–46.

[34] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 267–278. https://doi.org/10.1145/1555754.1555789

[35] Bernard Marr. 2019. How much data do we create every day? the mind-blowing stats everyone should read. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=7461a6c060ba

[36] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[37] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[38] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. DeepCache: A Deep Learning Based Framework For Content Caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML* (Budapest, Hungary) *(NetAI'18)*. Association for Computing Machinery, New York, NY, USA, 48–53. https://doi.org/10.1145/3229543.3229555

[39] Mike Owens. 2006. *The definitive guide to SQLite*. Apress.

[40] Reena Panda, Christopher Erb, Michael Lebeane, Jee Ho Ryoo, and Lizy Kurian John. 2015. Performance characterization of modern databases on out-of-order cpus. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 114–121.

[41] Pedro Pedreira, Chris Croswhite, and Luis Bona. 2016. Cubrick: Indexing Millions of Records per Second for Interactive Analytics. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1305–1316.

[42] 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*. 1203–1216.

[43] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 407–420.

[44] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. 1998. Performance of Database Workloads on Shared-Memory Systems with out-of-Order Processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VIII)*. Association for Computing Machinery, New York, NY, USA, 307–318. https://doi.org/10.1145/291069.291067

[45] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. 2021. Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 600–615. https://doi.org/10.1145/3445814.3446723

[46] B.R. Rau. 1988. Cydra 5 directed dataflow architecture. In *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*. 106–113. https://doi.org/10.1109/CMPCON.1988.4840

[47] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. 1989. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *Computer* 22, 1 (1989), 12–35.

[48] Richard M Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (1978), 63–72.

[49] Karthikeyan Sankaralingam, Tony Nowatzki, Vinay Gangadhar, Preyas Shah, Michael Davies, William Galliher, Ziliang Guo, Jitu Khare, Deepak Vijay, Poly Palamuttam, et al. 2022. The Mozart Reuse Exposed Dataflow Processor for AI and Beyond. (2022).

[50] Colin Schmidt and Adam Izraelevitz. 2015. A fast parameterized sha3 accelerator. In *tech. rep.* EECS Department, University of California.

[51] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 97–108.

[52] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[53] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2016. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *Commun. ACM* 59, 9 (aug 2016), 88–97. https://doi.org/10.1145/2975159

[54] Utku Sirin and Anastasia Ailamaki. 2019. Micro-architectural analysis of OLAP: limitations and opportunities. *arXiv preprint arXiv:1908.04718* (2019).

[55] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data*. 387–402.

[56] Akshitha Sriraman and Abhishek Dhanotia. 2020. *Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale*. Association for Computing Machinery, New York, NY, USA, 733–750. https://doi.org/10.1145/3373376.3378450

[57] Tiffany Trader. 2022. AMD's Genoa CPUs Offer Up to 96 5nm Cores Across 12 Chiplets. https://www.hpcwire.com/2022/11/10/amds-4th-gen-epyc-genoa-96-5nm-cores-across-12-compute-chiplets/

[58] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[59] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[60] Deepak Vohra. 2016. Apache parquet. In *Practical Hadoop Ecosystem*. Springer, 325–335.

[61] Michael Widenius and David Axmark. 2002. *MySQL reference manual: documentation from the source.* " O'Reilly Media, Inc.".

[62] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[63] Lisa Wu, Raymond J Barker, Martha A Kim, and Kenneth A Ross. 2013. Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 249–260.

[64] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. 2014. Q100: The architecture and design of a database processing unit. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 255–268.

[65] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. Flexpushdowndb: Hybrid pushdown and caching in a cloud DBMS. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2101–2113.

[66] Ahmad Yasin, Yosi Ben-Asher, and Avi Mendelson. 2014. Deep-dive analysis of the data analytics workload in cloudsuite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 202–211.