SwissBox: An Architecture for Data Processing Appliances

G. Alonso D. Kossmann T. Roscoe
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
www.systems.ethz.ch

ABSTRACT

Database appliances offer fully integrated hardware, storage, operating system, database, and related software in a single package. Database appliances have a relatively long history but the advent of multicore architectures and cloud computing have facilitated and accelerated the demand for such integrated solutions. Database appliances represent a significant departure from the architecture of traditional systems mainly because of the cross layer optimizations that are possible. In this paper we describe *SwissBox*, an architecture for data processing appliances being developed at the Systems Group of ETH Zurich. SwissBox combines a number of innovations at all system levels – from customized hardware (FPGAs), an operating systems that treats multicore machines as distributed systems (Barrelfish), to an elastic storage manager that takes advantage of both muticores and clusters (Crescando)– to provide a completely new platform for system development and database research. In the paper we motivate the architecture with several use cases and discuss each one of its components in detail, pointing out the challenges to solve and the advantages that cross-layer optimizations can bring in practice.

1. INTRODUCTION

Database appliances optimize the design, operation, and maintenance costs of data processing solutions by integrating all the components into a single system. Thus, an appliance typically uses customized hardware and storage, an operating system tailored to the application, and software highly optimized to the underlying platform. The advantage of such an approach is that cross-layer optimization and a high degree of customization can significantly improve the behavior and characteristics of the system over what is possible using independent, off-the-shelf components.

The recent interest in appliances is the result of a combination of factors. First, applications such as cloud computing and social networks have raised the bar for performance and scalability requirements. Second, existing cluster-based solutions implementing multi-tier architectures have proven to be both costly and difficult to maintain, opening the door for tightly integrated, closed systems maintained by the vendor. Third, the advent of multicore and fast networks allows enormous processing capacity to be packaged in just such a box. Last but not least, the enormous cost of today's infrastructure gives users plenty of motivation to consider customized solutions rather than using general purpose systems, creating the opportunity for highly tailored approaches focused on narrow vertical markets, or even on single use cases in large enough systems.

These trends, combined with the requirements from a number of real use cases, are the motivation for *SwissBox*: an architecture for data processing appliances we are developing in the Systems Group at ETH Zurich. SwissBox has two broad goals. First, we are addressing the challenges in engineering new appliances in the face of diverse and rapidly changing hardware and workloads.

Second, we are exploring the full ramifications of the freedom granted by the appliance model to re-architect the entire data processing stack from the hardware up to the application.

1.1 Background: Database appliances

The concept of a database appliance, broadly understood, has a long history. There are, however, important differences between current and past approaches to building appliances. Today's appliances use common off-the-shelf components (blade servers, operating systems, database engines, etc.) combined into a customized architecture. In contrast, previous, appliances like the *Gamma* database machine [4] were built using specialized hardware, with the consequence that their performance could be soon outstripped by rapidly advancing commodity systems. To illustrate this point, we briefly discuss three representative modern appliances: SAP's *Business Warehouse Accelerator*, the Netezza *TwinFin*, and Oracle's *Exadata*.

SAP's Business Warehouse Accelerator (BWA) speeds up access to a data warehouse by exploiting column stores, indexes in main memory, and multicore data processing. It consists of an array of computing blades plus storage nodes connected through a high-speed network switch. The data is read from the original database and reorganized into a star schema, reformatted as a column store, indexed, compressed, and written to the storage nodes. Query processing is done in the main memory of independent cores using on-the-fly aggregation, with the indexes also kept in main memory.

Oracle's Exadata combines "intelligent storage nodes" and database processing nodes in a shared disk configuration (based on Oracle RAC) connected by Infiniband. Exadata pushes relational operators like projections and join filtering to the storage nodes, thus reducing I/O overhead and network traffic. This is implemented with a function shipping module that complements the data shipping capability of the (conventional) database engine. To further reduce access latency, Exadata uses a Flash-based cache between the storage and the database nodes for hotspot data.

Netezza's TwinFin is based on a grid of "S-Blades" with a redundant front-end node. The grid is based on a custom IP-based

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

network fabric. Each S-Blade is a multicore processor where each core has an associated FPGA and disk drive. Data is streamed from the disks to the cores, with the FPGAs acting as filters which decompress the data and also execute relational operators such as selection and projection. In this way, an important part of the SQL processing is offloaded from the CPUs. TwinFin does not distinguish storage and data processing nodes, allowing for tighter integration of layers and, presumably, more optimized query execution.

Common to all these platforms is the combination of off-the-shelf components and a customized architecture that also includes specialized hardware. SwissBox is built in a similar fashion, but aims to explore in general the many cross-layer optimizations and alternative data processing techniques that become feasible in these new platforms.

1.2 Background: Use cases

SwissBox is motivated by real use cases provided by our industrial partners in the Enterprise Computing Center at ETH Zurich (`www.ecc.ethz.ch`). These emphasize the limitations of existing systems and suggest that a custom appliance could be a better approach to address them. Here, we focus on two such cases that underline the importance of cross layer optimizations.

The first is from Amadeus, the leading airline reservation provider. Amadeus acts as a cloud provider to the travel industry. Airlines, airports, travel agencies, and national authorities access reservation information through data services. Consequently, all queries have strict response time requirements (less than 2 seconds), and updates must also be propagated under tight time constraints. These guarantees must be maintained even under peak load (e.g., when a storm closes multiple airports in a region and a large wave of cancellations, re-bookings, and flight changes take place within a very short period). Furthermore, the query load does not consist solely of exact match queries: there is an increasing need for supporting complex queries under the same latency guarantees (for example, queries related to security).

As we have shown [16], the combination of tight latency guarantees and wide variety of loads cannot be handled with existing engines. The approach of materializing a view and using customized indexes for each data service quickly reaches a limit on scalability, not to mention the excessive maintenance cost associated with the full set of views and indexes necessary in a system of this size.

The challenge of the Amadeus use-case is combining OLTP and OLAP in a single system under very tight response time constraints. The requirements are as follows: deterministic and predictable behavior under a range of query complexity and update loads without the maintenance effort of complex tuning parameters, scaling to a large number of processing nodes, and enough reliability for continuous operation.

Existing commercial appliances address some, but not all, of these. All aim at reducing administration costs by removing tuning options: BWA automates schema organization and index creation, while TwinFin simply does without them. However, none of the commercial appliances we discuss provide, e.g., predictable response times or support for heavy update loads.

The second use case comes from the banking industry, and arises from the combination of increasing volumes of automated trading and the increasing complexity of regulations governing such trades. Algorithmic trading results in a high volume trade stream where the latency between placing the order and the order being executed is a key factor in the potential earnings. Existing systems can neither cope with the volume nor process the data fast enough to keep the latency to an acceptable minimum [13]. The need to inspect these real time streams for breaches of regulations, unauthorized trades, and violations of risk thresholds is a daunting but critical task.

The biggest challenge here is processing large volumes of data under tight latency constraints, with the processing involved becoming increasingly complex and sophisticated over time. The requirements we draw from the banking use case are: wire-speed data processing, low latency responses, dealing with data high volumes in real time, and scaling in both bandwidth and query complexity.

As above, commercial appliances at best address these requirements partially. Both TwinFin and Exadata emphasize the use of specialized networks to speed up operations, as the network quickly becomes the main bottleneck in such systems. Both appliances also move some processing to the storage layer (Exadata) or to network data path (TwinFin) to reduce the amount of data rate through the CPUs. However, none of these appliances support on-the-fly processing of data streams and they only provide hardware offload for simple operations.

1.3 Contributions

Motivated by these uses cases and the lack of a suitable solution, in this paper we outline SwissBox. SwissBox is a blueprint for data processing appliances that combines several innovative ideas into a single, tightly integrated platform. A key design principle of SwissBox is that extensive cross-layer optimizations (particularly across more than two layers) are a key high-level technique for meeting the requirements of modern application scenarios.

The importance of such techniques have been mentioned before. For instance, as part of the data stream processing carried out along complex and geographically distributed computational trees (an idea best captured by the Berkeley HiFi project [6]). In other areas of systems research, cross-layer techniques have also been the motivation for new designs of operating system [5] and proposals for changes to the Internet Architecture [11].

The appliance model allows cross-layer optimization to be applied to an entire stack within a single box. As such, the appliance model offers tremendous potential for gains from such optimizations. The corresponding challenge, however, is that essentially all layers of these systems need to be redesigned to both adapt them to the new hardware platforms and to open them up for a better interaction across layers, if their full potential is to be realized. The optimizations we propose in SwissBox represent an important step forward in the co-design of all systems layers.

In the short term, SwissBox is a practical way to address the challenges of predictability and scaleout presented by our use cases, in a way simply not possible with traditional database engines or cloud infrastructures. Longer term, SwissBox is a vehicle for exploring the wider possibilities opened up by the appliance concept. The model of custom configurations of COTS hardware within a closed appliance allows us to optimize across many layers in the software and hardware stack, by adopting novel designs of query processor, operating system, storage manager, hardware acceleration, and interconnect. We see SwissBox as a first step in this direction.

In the rest of the paper we describe our provisional design for the SwissBox architecture, and discuss lessons learned and research directions arising from our work so far.

2. SWISSBOX

SwissBox is a modular architecture for building scalable data processing appliances with predictable performance. The architecture consists of a flexible model of the underlying hardware, together with a functional decomposition of software into layers (though, as with network protocol stacks, this does not necessarily imply a layered implementation).

Many of the design decisions in SwissBox have been made both

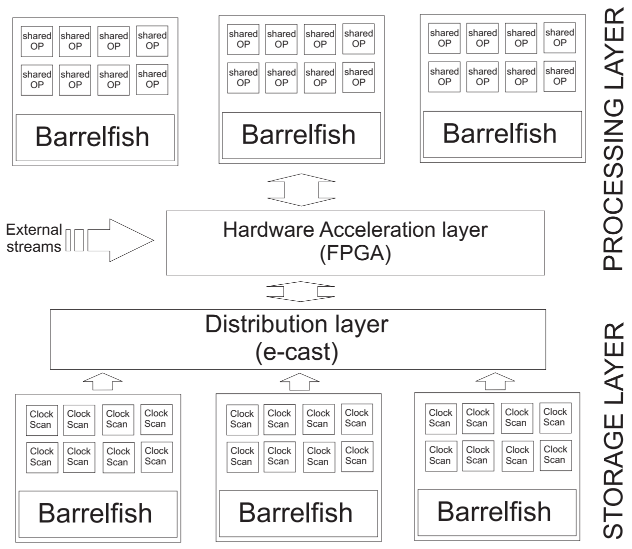

Figure 1: The architecture of SwissBox

to take advantage of current hardware and also to facilitate the adoption of new hardware developments as they become available. A SwissBox appliance consists of a cluster of commodity multicore computational nodes, each of which has multiple network interfaces attached to a scalable interconnect fabric. In turn, some of these interfaces will include programmable FPGAs on the datapath. SwissBox is agnostic as to the processor core types, and we expect a mix of general-purpose cores, GPU-like processors, and additional FPGAs to be used depending on the configuration. A promising candidate for the fabric is a 10 Gigabit OpenFlow [14] switch, which would provide plenty of opportunities for optimization across routing and forwarding layers inside the appliance, but Infiniband is a viable alternative. The fabric also has multiple external network ports for connecting the appliance to client machines.

The software architecture of SwissBox is shown in Figure 1. As in BWA and Exadata, SwissBox distinguishes between storage and data processing nodes. Storage nodes are multicore machines which implement the storage manager completely in main memory, removing the need for block-granularity operations and making it easier to adopt byte-addressable persistent storage technology such as Phase Change Memory as it becomes available. As in Exadata and TwinFin, storage nodes can themselves execute query operators close to the data. Reliability and availability is achieved by replicating the data across different nodes, while data partitioning provides scaling. The storage nodes also provide the basis for predictable performance by delivering tight guarantees on the time needed to access any data. Data processing nodes are also multicore machines configured as a cloud of dynamically provisioned resources. These nodes run complex query operators shared among concurrent queries and are deployed at the level of cores.

The layers in the software stack are described in detail in the sections which follow. At the lowest level, the Barrelfish research operating system [2] manages heterogeneous multicore machines and provides scheduling, job placement, interprocess communication, and device management to upper layer software. The *storage layer* is a distributed version of Crescando [16] (represented as a clock scan in each core in Figure 1), a main memory storage manager with precise latency guarantees that supports basic projection and selection as well as exact match and range queries.

The storage nodes are coordinated by a *distribution layer* using a novel agreement protocol call *e-cast* to enforce delivery and ordering guarantees while supporting dynamic reconfiguration. Between the distribution layer and the data processing nodes, a *hardware acceleration layer* uses FPGAs to pre-process data streams, building on our previous results in this area [17, 13]. Finally, the *data processing layer* implements high-level operators (join, sort, group-by, etc.) and scales using techniques such as result sharing across queries and orthogonal deployment of operators over cores.

3. OPERATING SYSTEM LAYER

An OS for SwissBox faces several key challenges.

One is traditional: there has always existed a tension between operating systems and database management systems (see, for example, Gray [9]) over resource management. To somewhat oversimplify, an OS is usually designed to multiplex the machine between processes based on some global policy, and present an abstract "virtual machine" interface to applications which hides the details of resource management. This is generally useful, but for databases it obstructs the application from making optimizations based on resource availability, and denies it the mechanisms to internally manage and multiplex the resources allocated to it.

Other challenges are specific to neither databases nor appliances, but are consequences of recent hardware trends. The advent of multicore computing, where the number of cores on a die now roughly follows Moore's law, has lead to scaling challenges in conventional OSes like Windows or Linux. Such systems are based on coherent shared memory, and suffer in performance as core count increases due to contention for locks, and (more significantly) the simple cost of moving cache lines between cores and between packages. Such scaling effects can be, and are, mitigated in traditional OS kernels, but at a huge cost in software engineering [2]. Worse, these expensive measures are typically rendered obsolete by further advances in hardware, requiring further re-engineering. for appliances.

Furthermore, traditional OS structures are simply unable to integrate heterogeneous processing cores (instruction set variants, GPUs, etc.) within a single management framework, nor will they function on future machines without coherent caches or shared memory, which are being discussed by chip vendors [10].

Commercial appliances use off-the-shelf OS kernels, which the vendors then specialize to their hardware platform. This takes advantage of the freedom given in the appliance model to optimize layers of software without undue concern for strict compatibility, and using commodity software dramatically reduces time-to-market. However, it does not fully meet the emerging challenges outlined above, most of which arise from the basic architecture of the OS. In addition, the approach does not exploit the opportunities to rethink the software stack across layers.

In contrast, the OS in each node of SwissBox is Barrelfish [2], a research operating system specifically designed to manage heterogeneous, multicore environments. Barrelfish is a "multikernel": an OS structured as a distributed system whose cores communicate exclusively via message-passing. This provides a sound foundation for scalability in the future by completely replacing the use of shared data structures with replication and partitioning techniques. By separating efficient messaging from core OS functionality, Barrelfish is less reliant on short-lived hardware-specific optimizations for multicore performance. As a distributed system, Barrelfish also easily handles heterogeneous processing elements and machines without cache coherence.

We also expect Barrelfish to be a better fit for data processing applications. On each core, the OS is structured as an Exokernel [5], minimizing resource allocation policy in the kernel and delegating management of resources to application as much as possible. Barrelfish extends this philosophy to the multicore case through the use of distributed capabilities. The resulting OS retains responsibility for securely multiplexing resources among applications, but presents them to applications in an “unabstracted” form, together with efficient mechanisms to support the application managing these resources internally. To take two examples very relevant to databases, CPU cores are allocated to an application through upcalls, rather than transparently resuming a process, allowing fine-grained and efficient application-level scheduling of concurrent activities, and page faults are vectored back to applications, which maintain their own page tables, enabling flexible application management of physical memory.

Beyond this, the use of a flexible research OS within SwissBox opens up the exciting possibility of *OS-Database codesign*: architecting both the operating system and the data processing application at the same time. Such opportunities are rare, and the design of the appropriate interface between an OS and the data storage and processing layers in SwissBox is an exciting research direction. For instance, the SwissBox storage layer knows well the read/write patterns it generates, and we are exploring how to pass that information to the OS so that the OS can made intelligent decisions rather than getting on the way or having to “second guess” the application. Similarly, from those patterns the OS could derive copy on write optimizations to facilitate recovery and seamless integration of the main memory storage layer with a file system.

4. STORAGE LAYER

The storage layer in SwissBox is based on Crescando, a data management system designed for the Amadeus use case [16]. In SwissBox, we use a distributed version of Crescando as the storage layer because of its predictable performance and easy scalability. It also establishes the basis for exploring completely new database architectures that provide high consistency, exploit the availability of main memory, while still enabling the levels of elasticity and scalability needed in large applications.

In SwissBox, the interface to the storage manager is at the level of tuples rather than at the level of disk/memory blocks. Similarly, rather than simple get and puts over blocks, the storage is accessed via predicates over tuples. This greatly facilitates pushing part of the query processing down to the storage layer and makes the storage manager an integral part of the data processing system not just an abstraction over disk storage. This design choice is crucial to make SwissBox open to further hardware developments and to provide the same interface regarding of the hardware (or hardware layers) behind the storage manager.

Briefly described, Crescando works as a collection of data processing units at the core level. The storage manager works entirely in main memory, using clusters of machines for scalability and replication for fault tolerance. Each core continuously scans its main memory (hence the name *clock scan*), using an update cursor and a read cursor where queries and updates are attached in every cycle. Instead of indexing the data, Crescando dynamically indexes the queries to speed up checking each record against all outstanding queries. The latency guarantees of Crescando are determined by the time it takes to complete a scan. In current hardware, Crescando can scan 1.5 GBytes per core in less than a second, answering thousands of queries per cycle and core, even under a heavy update load. Within a scan, Crescando maintains the equivalent of snapshot isolation, with consistency across the whole system being maintained by *e-cast*, an agreement protocol specifically developed for Crescando (see below).

Crescando partitions the data into segments that are placed into the local memory of a single core. The segments can be replicated across cores and/or across machines, depending on the configuration chosen. For availability, Crescando maintains several copies of each segment across different machines so that if one fails, the segment is still available at a different node. Recovery in Crescando only requires to place the corresponding data in the memory of the recovered machine.

To understand the advantages of Crescando, it is important to understand how it differs from a conventional engine. For single queries over indexed attributes, Crescando is much slower than a conventional engine. However, as soon as a query requires a full table scan or there are updates, Crescando performs much better and with more stability than conventional engines. In other words, Crescando trades-off single, one at a time query performance for predictability (in Crescando all queries and updates are guarantee to complete within a scan period) and throughput (Crescando is optimized for running thousands of queries in parallel).

When compared to existing appliances, the storage layer of SwissBox offers several advantages.

As in Exadata, Crescando is an intelligent storage layer that can process selection and projection predicates, as well as exact match and range queries. This eliminates a lot of unnecessary traffic from the network and offers the first opportunity within SwissBox for pushing down operators close to the source. Unlike Exadata, in SwissBox we do not need an extra function shipping module or extra code at the storage layer. The storage layer is built directly as a query processor that can be easily tuned to match any latency/throughput requirements with two simple parameters (the size of a scan and the level of replication).

Like in SAP’s BWA, the storage layer of SwissBox processes data in main memory, avoiding the overhead of data transfer from disk. In SwissBox, indexes are not used and the database administrator does not need to worry about identifying which ones are needed. This is an important point as one of the motivations for, e.g., SAP’s BWA is to provide performance in spite of the fact that users are likely to chose the wrong indexes and data organization (and this is why BWA reorganizes the data). In the future, the main memory approach of SwissBox is ideally suited to quickly adopt Phase Change Memory, thereby combining the advantages of in memory processing with persistent storage.

In terms of cross layer optimizations, the design of Crescando fits very well with the premises behind Barrelfish. Crescando can use the interfaces provided by Barrelfish to make runtime decisions on placement and scheduling. The data storage layer can also be combined with modern networking techniques such as RDMA (Remote Direct Memory Access) [7]. Through RDMA, a node can place data directly in the memory of another node without involving the operating system and minimizing copying of the data. This is a feature that is commonly available in high end networks like Infiniband (the network used in Oracle Exadata) but that is now available through special network cards also for Ethernet networks. Using RDMA, recovery of nodes, data reorganization, and dynamic creation of copies can be greatly accelerated over traditional approaches. For instance, there is some initial work on rethinking the processing of data joins using RDMA [8].

As an alternative design option, we are also exploring using a similar one-scan-per-core approach but on a column store (as in SAP’s BWA). Initial experiments indicate that column stores in main memory allow interesting and quite effective optimizations even if no disk access is involved. An intriguing idea that can be easily supported by the SwissBox storage layer is to store the data both row-wise and column-wise, routing queries to the most appropriate representation.

5. DISTRIBUTION LAYER

The distribution layer of SwissBox is one of its distinct features that differentiate it from existing appliances and that confers SwissBox very interesting properties. Such a layer is commonly found in cloud solutions, particularly in distributed key value stores, e.g., [1], but we are not aware of any database appliance using anything similar.

In SwissBox, this layer is implemented on the basis of a new agreement protocol, e-cast, that distributes reads and writes across the storage layer nodes. E-cast combines results from virtual synchrony [3] and state machine replication [12, 15] to provide a highly optimized protocol that maintains consistency across replicated, partitioned data. Aside of the performance and configuration advantages it offers, e-cast supports dynamic reorganization for both transparent fail-over and elasticity. Through these features, SwissBox is in a position to easily scale up and down its storage layer, something that is not possible as far as we are aware with existing appliances.

The distribution layer offers many opportunities for research and exploring new architectural designs. For instance, E-cast works across nodes of the storage layer. Barrelfish does something similar but across the cores of a multicore machine. As part of ongoing work, we are analyzing the characteristics of multicore machines as distributed systems and taking advantage of their special properties (e.g., messaging implemented as shared memory, synchronized clocks) to develop a suite of agreement protocols specially tailored to multicore machines. An interesting design option would be to interface e-cast and Barrelfish so that the two of them act as seamless extensions of each other. This is particularly interesting in SwissBox because both the storage management and data processing layers are organized in terms of independent units of execution allocated to one core. The resulting protocol would act as a hierarchical network that would allow SwissBox to treat a pool of multicore machines not as a series of independent nodes but as a pool of cores with the distance between them (the network overhead) as the parameter to use for query optimization.

The elasticity provided by e-cast will also play a crucial role in the automatic configuration of SwissBox. As pointed out above, a key parameter in tuning the response time of the storage manager is the size of a clock scan. The smaller the scan, the faster the storage layer but the more nodes are needed. With e-cast we have a way to easily reorganize the storage layer: consolidating the data in few nodes if the response time constraints allow it or expanding the storage layer across many nodes if the size of the scan needs to be reduced or the level of replication needs to be increased to meet the given performance requirements. E-cast provides here the logic to maintain consistency in the face of dynamic reconfigurations while techniques like RDMA provide the mechanisms for quickly migrating, copying, or reorganizing the independent work units allocated to cores.

6. HARDWARE ACCELERATION LAYER

Both the use case from the financial industry discussed above and the emphasis of commercial appliances in fast networks point to one of the major bottlenecks encountered today by cluster based solutions: the network. SwissBox is no exception in this regard. The problem is twofold. On the one hand, concurrent accesses compete for bandwidth and the network latency is higher than on a local disk. On the other hand, using a network card typically involves the operating system, additional data copies, and a reduced capacity to process data at the speed it arrives from the network.

In SwissBox we use an additional layer of FPGAs for processing the data streams as they travel from the storage layer to the data processing layer. The hardware acceleration layer offers a second tier to which operators and parts of queries can be pushed down. This layer is similar in functionality to that found in TwinFin and can be used for a variety of purposes: aggregation, decompression, dictionary translation and expansion, complex event detection, or filtering of tuples. The key advantage of this layer is that we have shown it can process data at wire speed, something that conventional network interfaces cannot do unless the number of packets from the network is drastically reduced [17].

An interesting aspect of the hardware acceleration layer is that the implementation of data processing algorithms in hardware requires very different techniques than those commonly used in CPU based systems. For instance, in [17] we have shown that the functionality of a hash table can be implemented in hardware with a pipelined circuit that can process multiple elements in parallel, reaching a much higher throughput than in a CPU based system. Such fundamental differences in the underlying algorithms raise questions about the ability to compile queries where operators will be placed in different tiers and one of those tiers involves hardware based operators. Both Exadata and TwinFin claim that they place operators on different layers (the intelligent storage of Exadata, or the FPGAs of TwinFin) although the query compilation process and the cost optimization functions that guide the query compilation are not obvious. From the available information, these systems place only simple operators such as selection, projection, and decompression in the lower layers. A question that we aim at answering as part of the evolution of SwissBox is whether more complex operators can also be pushed down to the lower layers and what are the cost models to use as part of the query optimization process [13]. To do this, an ongoing research effort around SwissBox is how to characterize data processing operators so that we can provide a uniform interface to those operators regardless of where they are located in the data processing hierarchy within an appliance.

7. DATA PROCESSING LAYER

The layers described so far can only execute relative simple queries. To support the full of SQL or even generic data processing operators written in other languages, SwissBox incorporates a data processing layer where operators for more complex queries are executed. These operators are allocated to cores as independent work units in nodes running Barrelfish (in a similar configuration as the storage nodes). The scalability and elasticity in the design comes from the fact that the operators can be deployed in arbitrary cores/machines (similar to TwinFin and BWA), with the placement controlled by the query plan optimization. Unlike in existing systems, these operators do not work one query at a time. They have been designed to be shared by many concurrent queries (in the thousands) and make heavy use of intermediate result sharing. The way this is achieved is by using the Crescando storage nodes –which also process queries concurrently– to label a stream of results with the id's of the queries for which the record is intended. These result sets are streamed to the corresponding data processing node (after filtering through the FPGA layer), which then performs the operation concurrently for all queries, separating the results only afterwards. This design increases the possible throughput considerably and also helps to reduce the data traffic across the system since records that are in the result set or intermediate result set of many queries are sent only once rather than once for each outstanding query.

Note that the data processing operators can be placed on data processing nodes or they can also be placed on the core of storage nodes. This gives SwissBox another degree of freedom for deploying highly optimized query plans. For instance, operators such as sort, group by, and aggregation can be easily placed on cores next to Crescando cores. The result is that there is no traffic over the network as, in many cases, the entire query can be pushed down to the storage layer. This design mirrors somewhat that of TwinFin in that it blurs the separation between data processing and storage nodes. Unlike TwinFin, however, we can eliminate the traffic over the network entirely for some queries and we can choose to place the operators in the data processing nodes instead for scalability and elasticity purposes.

The advantage of the data processing layer approach is that we can maintain the predictability of the system by ensuring the execution of a part of a query that runs on these nodes has a well defined upper bound. The overall cost of a query would then be the time to get the data from the storage layer (which is accurately defined thanks to the Crescando approach), the transmission overhead of the network if any, and the processing overhead of the high level SQL operators in the data processing nodes. We believe that the unique architecture of SwissBox allows to place tight bounds on these overheads and to capture these bounds with very few parameters. This opens up the possibility of tools for automatic configuration of all layers of SwissBox given a set of response time requirements. Note that such tools will make heavy use of the open interface of Barrelfish, the well defined tuning parameters of Crescando, and the elasticity of the data processing layer, with the hardware acceleration and the possibility of pushing down operators close to the storage layer providing additional leverage to address the problem.

8. CONCLUSIONS

SwissBox is intended both as a data appliance to be deployed in real settings and as a vehicle for research to enable the complete redesign of the software stack.

As an appliance, SwissBox has a number of unique features that make it very suitable to the use cases described earlier as well as in a wide range of other environments. In SwissBox, there is no locking or read-write contention at the storage layer, allowing us to support very high update rates without impacting the read rates. Moreover, all layers of the system but specially the storage manager –which is traditionally the main problem in this regard– provide predictable performance for both reads and writes. This allows us to build systems that by design can meet tight response time constraints.

When compared to key-value stores like Cassandra [1], SwissBox provides full data consistency, elasticity, and the ability to perform complex SQL queries, thereby establishing a completely new point in the design space. When compared to existing appliances, SwissBox's multi-query optimization strategies at all levels and the organization of work into independent units mapped to cores provide a degree of flexibility and scalability that cannot be achieved with disk based systems. In SwissBox we can benefit from many of the same hardware optimizations used in existing appliances (e.g., SSD storage, FPGA processing) but we can also exploit many other cross-layer optimizations that are not possible in today's appliances thanks to the use of Barrelfish as the underlying operating system.

As a research platform, SwissBox gives us the opportunity to completely redesign the data processing stack from the ground up, exploring at the same time how to best take advantage of new developments in hardware. This is a rather urgent matter given the pace at which key elements of the data processing infrastructure are evolving. That SwissBox is an appliance makes it possible to apply cross-layer optimizations within a single box, thereby opening up the possibility of taking these cross-layers optimizations much further than it has been possible in distributed platforms. At the same time, it will allow us to rethink the interfaces and role of the different layers of a data processing system in terms of meeting new requirements like predictable performance or elasticity. Our intention is to make SWissBox open source -even the hardware architecture- to provide an open platform for experimentation and education where new data processing techniques can be tested and compared free from the limitations and legacy constraints of existing database engines.

9. REFERENCES

[1] *http://cassandra.apache.org/*.

[2] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, pages 29–44, 2009.

[3] K. Birman. A History of the Virtual Synchrony Model. Technical report, Cornell University, 2009. http://www.cs.cornell.edu/ken/History.pdf.

[4] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. SOSP*, pages 251–266, December 1995.

[6] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. W. 0002, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*, pages 290–304, 2005.

[7] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *ICDCS*, pages 553–560, 2009.

[8] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. In *ICDCS*, pages 283–292, 2010.

[9] J. Gray. Notes on database operating systems. In Bayer et. al., editor, *Operating Systems, an Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

[10] J. Howard and et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International Solid-State Circuits Conference*, pages 108–109, Feb. 2010.

[11] R. R. Kompella, A. Greenberg, J. Rexford, A. C. Snoeren, and J. Yates. Cross-layer visibility as a service. In *In Proc. IV HotNets Workshop*, 2005.

[12] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 6(2), 1984.

[13] R. Müller, J. Teubner, and G. Alonso. Streams on Wires - A Query Compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.

[14] OpenFlow Consortium. Openflow. `www.openflow.org`, September 2010.

[15] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(299), 1990.

[16] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1):706–717, 2009.

[17] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *PVLDB*, 3(1), 2010.