FlowPuter: A Cluster Architecture Unifying Switch, Server and Storage Processing

Alfred V. Aho, Angelos D. Keromytis, Vishal Misra, Jason Nieh, Kenneth A. Ross, Yechiam Yemini
Department of Computer Science
Columbia University
{aho,angelos,misra,nieh,kar,yemini}@cs.columbia.edu

***Abstract*— We present a novel cluster architecture that unifies switch, server and storage processing to achieve a level of price-performance and simplicity of application development not achievable with current architectures. Our architecture takes advantage of the increasing disparity between storage capacity, network switching on the one hand, and processing power of modern processors and architectures on the other. We propose the use of Network Processors (NPUs), which can apply simple classify/act/forward operations on data packets at wire speeds, to split processing of operations such as complex database queries across a network. We quantify the theoretical benefits of such an architecture over traditional server-cluster approaches using warehouse database queries as a motivating application. We also discuss the challenges such an architecture presents to programming language design and implementation, performance analysis, and security.**

KeyWords: Dataflow processing, Network Processors, Data Warehouses, Security, Compilers, Performance.

I. Introduction

Traditional computing assumes that I/O speeds are substantially slower than CPU-memory speed. Emerging network and storage technologies have been stretching this assumption to its limits. Optical wire-speeds already range in the 10-40 Gbps, on par with CPU-memory bandwidth, while hard-disk drives are delivering 2.5 Gbps. With both wire-speed and storage density technologies improving faster than silicon speeds, it is not unreasonable to assume that raw I/O speeds can continue to outgrow CPU-memory speeds this decade. This speed inversion will have a profound impact on computing.

Despite these dramatic improvements in raw I/O speeds, traditional computing approaches will not be able to take advantage of these advances in hardware technology. Current applications are often structured assuming fast random access to data. While raw bandwidth is increasing at a dramatic rate, access times are not. Consider disk storage technology as an example. Disk storage density is increasing at a rate of 100 percent a year, much faster than Moore's law. Even if disk rotational speed remains constant, this improvement in density can translate into an equivalent percentage increase in sequential throughput rate since there is that much more data that can be accessed in each disk rotation. However, disk access times are only increasing at a rate of about 10 percent a year. As a result, random I/O access patterns that each incur disk access times will continue to be relatively slow while sequential I/O access speeds outstrip CPU-memory speeds.

Part of this research was funded by NSF CAREER award ANI-0238299 and NSF Contracts CCR-TC-0208972 and IIS-01-20939.

Growth in storage capacity and in respective content flows has been stimulating commensurable growth in the sizes of data sets. Sensors are producing an enormous amount of data that can be manipulated and processed, multimedia documents are many times richer and larger than their plain text counterparts, streaming media is becoming increasingly common, high-definition imaging and visualization is increasingly common for medical and scientific applications, and large databases and data warehousing applications are a multi-billion dollar business. With the first multi-TB drives expected in 2005 and with numerous application scenarios requiring processing data in real-time with its flow to storage, processing multi-TB at 10 Gbps or more is not an unrealistic expectation for applications needs by the end of the decade.

The confluence of changing hardware technology and increasing I/O application demands requires a fundamental shift in computing technology. We propose Flow Computing, a new approach to computing that leverages the continuing rapid improvements in network and storage speeds. Our proposed approach uses a switch-like architecture to circulate data repeatedly through a high-speed network to and from disks. Primary computations on data flowing through the switch architecture classify and filter data in a highly parallelized and pipelined manner to provide a high cycle budget to process a high-speed sequential flow of structured data units. To provide the necessary processing cycles, the architecture utilizes chip-level parallelism provided by generic network processors (NPUs) or FPGA-based approaches. Network flows are then multicast and routed among NPUs to enable processing by multiple applications. We call such such a hardware/software organization a *flowputer.*

Emerging server, switch and storage architectures have been converging to support flowputer scalability. Specifically, dense-blade servers are already organized into a switch-like architecture to provide scalable cycle budget and switching capacity to handle network flows; switches, likewise, have been incorporating higher-layer processing functions and using NPUs/FPGAs to provide the cycle-budget, with some switches already integrating server blades; both could be used to support Flowputing software architectures. There is thus a need to develop better Flowputing software technologies that can exploit and guide these emerging hardware architectures in accomplishing cycle-efficient, software-efficient and flow-

efficient flow processing.

In the remainder of this paper, we explore the vision of the flowputer architecture in more detail, using data warehousing as a motivating example, explained in detail in Section II. The warehouse example is generic and its considerations generalize easily to other I/O-intensive processing application scenarios, including streaming media and image processing, database processing, XML document-store processing, and speech processing. In particular, we discuss the potential benefits to be had from such an architecture (Sections III and IV), and the challenges from the security, programming language, and performance analysis standpoints (Sections V, VI, and VII respectively).

II. Data Warehousing using Flow Computing

A large organization can gather all of its data into a single very large database for the purpose of analyzing various aspects of its enterprise. This activity is called *data warehousing*, and there is currently a multi-billion dollar market for data warehousing systems. The value of the data warehouse is that it can provide access to detailed information about current activities that enable the organization to adjust its behavior. The data warehouse is optimized for fast query performance, with updates happening in batches. Since most queries touch large quantities of data, and since answers are not required in real-time, the overall performance metric is typically query throughput.

Most current data warehousing systems are based on large clusters of commodity processors [1]. Such systems have two potential weaknesses. The first weakness is how the system scales with increases in the query workload, given a fixed large data set. Because disks and data are highly distributed, it is hard to share work (such as I/O) between queries. As a result, the hardware resources, from the CPU to the disks, need to scale in proportion to the aggregate query workload.

The second weakness is manageability. Query processing and optimization is a very complex process, and query optimizers for commercial data warehousing systems are extremely intricate pieces of software. Query optimization depends on a very large number of parameters, including (potentially for each column of each table) information about physical layout, index availability, replication, etc. The resulting complexity makes managing such a data warehouse labor intensive. Despite recent efforts such as the SQL server index selection wizard [8], choosing good parameters for a large database installation remains an art that requires highly skilled administrators. The management costs of database installations have begun to dominate the overall cost of ownership [11].

An alternative, visionary perspective was provided as early as the late 1980s by the Datacycle system [12], [5], [6], which resembles a flowputer. A data pump cycles through the entire database, transmitting it over a network. Data processors tap into the network through a custom-built VLSI interface. The interface is capable of performing filtering operations in hardware, so that processors see only data relevant to their queries. Many processors, each with their own interface, could potentially tap into the same network, thus sharing a single data flow for different operations.

Compared with a cluster of commodity processors, the Datacycle system scales better with respect to the query workload. As the query workload increases, one needs only to add processors and interfaces to the shared network. The underlying data storage and pump infrastructure does not need to get bigger. Further, as recognized by the Datacycle architects, the overall system is easier to manage. Indexes are not used, since all a processor has to do is set a filter for the appropriate data and it will be captured as it flows past on the network. Similarly, query processors do not have to concern themselves with issues of physical layout of data on disks. As a result, there are far fewer parameters, and management of the whole system is simpler.

Like much of the “database machine” research of that period [14], the Datacycle architecture suffered from a fatal flaw: it used special-purpose fabricated hardware. As commodity processors rapidly improved in performance, any performance advantages provided by the special hardware quickly evaporated, unless expensive design and fabrication was undertaken with each new VLSI technology.

We argue that the time has come to revisit this kind of architecture, because commodity switches now have the capability to provide the filtering capabilities provided by the VLSI interfaces in the Datacycle system. As a result, a “commodity flowputer” can provide all of the advantages that the Datacycle architecture promised, without its fatal flaw.

In fact, we believe that a flowputer can go beyond the capabilities of the Datacycle architecture in several important ways. Ironically, the Datacycle system predates the entire field of data warehousing, and so people have not studied in depth how complex query processing could utilize such an architecture.

III. The FlowPuter Architecture

Emerging clusters redefine computing in terms of stack processing, and redefine switching in terms of classify/act/forward operations that use any packet header (and even contents) as filters. Furthermore, they redefine storage access in terms of routing I/O accesses within virtual storage spaces. There is, therefore, an emerging confluent semantic of computing, switching and I/O whereby data, tagged by multiple layers, flows through engines that apply classify/act/forward processing to these tags. We call this unifying semantic flow-processing and the engine that handles it a *flowputer.* A flowputer that processes a database stream views the task as classifying and processing nested tags–whether these are implicit in the structure of TCP/IP headers, explicit in the HTTP headers, or defined by SOAP and other XML libraries, and represent fields of a database record, or a virtualization mapping that indicates storage routing. A flowputer manipulates flows of such tagged data at all layers using a common platform.

The hardware architecture of a flowputer is depicted in Figure 1. Flowputer blades incorporate powerful Network Processor (NPU) hardware to provide the cycle budget needed

for wire-speed processing of tagged flows; network flows are delivered to/from a flowputer blade through interface hardware. Flowputer blades are interconnected by a fast switching fabric.

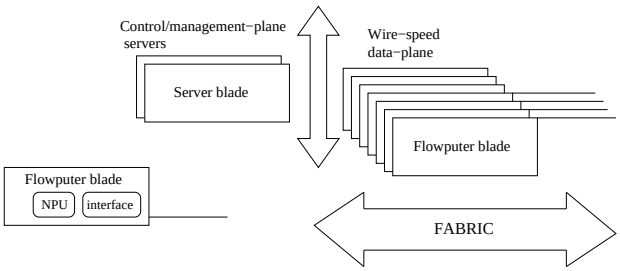

Fig. 1. **FlowPuter hardware architecture.**

The software architecture of a flowputer is organized along a data-plane and control/management-plane separation. The control-plane software has a standard server architecture; its main function is to manage and configure tag-processing functions of the data-plane (*e.g.*, maintain and load tag libraries, configure routing relationships among data-plane modules, configure shared data of data-plane modules, *etc.*). The data-plane software consists of a light-weight run-time execution environment to process tagged flows through classify/act/forward modules and channels.

In a typical scenario, a database flow arrives at a flowputer blade through a network connection. The flowputer processes the network stack to direct the database flow to respective processing channels. The flow may be multicast to multiple applications executing at different flowputer blades. Each such application may process the flow through multiple channels organized as pipelined and/or parallel classify/act/forward operations. The database records may be filtered through channels that project or select records as defined by an application. Thus, a large number of simultaneous queries may be applied concurrently to the same database.

The flowputer accomplishes significant scaling efficiency over current clusters. This efficiency arises from several factors:

1) The flow processing model eliminates the typical overheads of synchronizing I/O events with process events. The flowputer processes are optimized to service flow events and involve minimal overheads.
2) The flowputer hardware uses specialized classification/forwarding elements to dramatically accelerate the classify/forward parts of processing; thus a tag may be classified in a single processor cycle *vs.* hundreds of cycles needed to process it on a CPU.
3) The flowputer exploits the much larger cycle budget provided by NPUs, which use pipelined/parallel micro-engines.
4) The flowputer enables a scalable number of concurrent applications to share storage I/O access. For example, thousands of database queries may be processing the database concurrently by sharing its flow.

These different scaling efficiencies create a fertile ground for superior performance over farm architecture. Table I contrasts the scaling features of a flowputer against a more traditional server-cluster in processing a warehouse. We consider four scenarios (rows 2-5) through which the size of the warehouse *(c2)* and the number of applications that process it *(c4)* grow, while the processing time to complete these applications *(c3)* shrinks. The table makes the following assumptions:

- The NPU offers 8 pipelines of 4 stages, operating at 250 Mhz, for a total cycle budget of 8 Ghz, contrasted with 1 Ghz CPUs of the server-cluster. Both the server-cluster and the flowputer are operated at 50% utilization to minimize queuing.
- The server-cluster CPUs requires 4,000 cycles to process a warehouse record (500 to classify, 500 for handling and 3k overheads) while the flowputer takes 1,000 cycles, by reducing the overheads and using fast classifiers/forwarders.
- The warehouse storage systems can be accessed over the network links at 1 Gb/sec per TB stored; when the applications demand storage access bandwidth exceeding the capacity of storage systems, the warehouse is replicated; replication and competition over storage access add overheads that increase the cycle budget required by 2% for each additional application. The server-cluster applications access storage randomly, retrieving 1 MB chunks per access; the flowputer uses sequential access to retrieve very large (>50 MB) blocks, increasing the effective bandwidth by minimizing the impact of seek time.

Under these assumptions, it is simple to compare the scalability features of these architectures:

1) Columns *c5* and *c6* contrast the required number of server-cluster CPUs *vs.* flowputer NPUs, illustrating their relative efficiency in scaling the cycle budget. Under scenario B, for example, the server-cluster requires 53 CPUs while a flowputer requires 2 NPUs. Even scenario D would require just 1-2 flowputers with 66 NPUs, contrasted with a server-cluster requiring hundreds of blades. The ratio #CPU/#NPU, measuring the cycle-budget scaling efficiency of the flowputer, grows from 3 to 51.
2) Columns *c7* and *c8* contrast the storage access bandwidth (BW) required by the two architectures. For example, under scenario C the flowputer requires access at 4 Gb/sec, entirely resulting from the need to move 4TB in 2 hours. In contrast, the server-cluster requires 196 Gb/sec, primarily resulting from the need to support the aggregate storage access needs of 20 applications.
3) Columns *c9* and *c10* contrast the number of replicated copies of the warehouse needed to support the access bandwidth of concurrent applications. A server-cluster would need to replicate the warehouse creating as many as 147 copies to handle scenario D. The flowputer does not require replication of the data.

IV. Using FlowPuter in Data Warehousing

We focus now on two novel uses of the switch infrastructure to process queries in ways not possible either with a conven-

c1	*c2*	*c3*	*c4*	*c5*	*c6*	*c7*	*c8*	*c9*	*c10*
Scenario	**Warehouse Size (TB)**	**Processing Time (hr)**	**# Apps**	**#CPU farm**	**#NPU Flwptr**	**BW Farm (Gb/sec)**	**BW Flwptr (Gb/sec)**	**#Rplca Farm**	**#Rplca Flwptr**
A	1	4	1	3	1	1.2	0.6	2	1
B	2	4	10	53	2	24	1	12	1
C	4	2	20	487	11	196	4	49	1
D	8	1	30	3334	66	1173	18	147	1

TABLE I
Scaling Effects For A Warehouse Processing Example.

tional database cluster, or with the basic Datacyle architecture. We then conclude this section by highlighting some additional research questions we plan to address.

A. Aggregation in the Switch

Assume a dataflow of records represented in a format that can be interpreted by the switch hardware. The switch can extract a database field and perform a filtering operation, so that records matching a condition are sent along an appropriate processing path. Cascades of filters can be programmed into a switch.

In addition to filters, some query processing can also be performed in the switch itself. For example, consider the following query:

```
SELECT US_State, Sum(sales)
FROM transaction-table
WHERE date > 01-01-2002
GROUP BY US_State
```

One way to perform this query is to filter the records on "date" and then send the matching records to a CPU for aggregation. An alternative way to execute this query is to compute the aggregates *in the switch*. Since there are 50 States in the US, the switch needs to allocate memory for 50 accumulators, which are initialized to zero. 50 filters are created, one per State. Records that match the date criterion are then run against these 50 filters. Since the filters are in associative memory, the filters on State take only one unit of time. When a State filter matches, the local accumulator is incremented by the value of "sales" in the current record.

The advantage of this alternative plan is that both network traffic and computational work is removed from the CPUs. The extra bandwidth and processing capacity can be used for other queries. Such a plan requires certain resources, namely memory for filters and accumulators. Modern switches have the capacity for millions of filters and accumulators. These resources must be allocated in a fashion that maximizes their benefits for overall query processing.

B. Dynamic Load Balancing

In a conventional clustered database system there is an extensive reliance on parallel processing among the nodes. In order to maximally utilize resources, query processing plans must have access to data distribution statistics. Even with such access, it is often very difficult to accurately estimate the size of intermediate results. As a result, the allocation of work among the nodes may be suboptimal, giving too much work to some, and too little to others, increasing the response time of the query. Even worse, the derivation of a query processing plan usually does not take into account what other queries may be running on each node. A single overburdened node can thus slow down the entire query. Shipping sub-queries from one node to another is often impractical due to the large data sizes.

A flowputer, on the other hand, sees a single unified input data stream, and can divide up the work dynamically based on current workload statistics. For example, suppose that we have four CPUs C1 through C4 processing an aggregate query. The switch divides the input into four disjoint streams, one per CPU, using a hash function H on the input data stream. Each CPU computes aggregates over its input. In a final step, the values from each of the four CPUs are combined to answer the query. Using four CPUs gives us a degree of parallelism, as well as four times the net bandwidth of a single CPU.

In a flowputer, we can adjust the hash function H dynamically. If C1 is getting more than its share of records due to some bias in the data distribution, we can modify H so that a smaller range of hash values gets mapped to C1, and a larger range gets mapped to other processors. Similarly, if C2 happens to be running slowly because of other queries in the system, the switch may dynamically increase the proportion of records mapped to C1, C3 and C4 by adjusting H. Note that adjustments due to data skew or machine overloads are handled in the same way, at query-processing time.

V. FLOWPUTER SECURITY

The flowputer architecture is, by nature, exposed to certain risks not present in a traditional warehouse. In particular, since the warehouse records are transmitted over a network and queries are evaluated in a distributed manner across a number of NPUs, we are concerned both about data confidentiality and integrity (*i.e.*, preventing an outsider from obtaining the data by placing a tap one of the network links), as well as access control (ensuring that legitimate users can access only data for which they are authorized). Currently, data warehouses are typically protected from outsiders by firewalls and physical measures; access control is performed by the database server, by restricting the database tables to which a query has access.

A. Data Security

To the extent that the entire flowputer can be kept physically and virtually isolated from the public network, similar measures against outside attackers will be equally (in)effective. However, when we consider a distributed enterprise, where each branch needs access to specific tables and records, we must treat the interconnecting links as part of the public network. The only currently-known practical security mechanism for protecting data confidentiality and integrity over such links is cryptography[1].

Although the current generation of cryptographic accelerators cannot sustain the flowputer's data rates, future versions that can accommodate data rates of 10 Gbps have been announced by different vendors. However, as is shown in [13], the limiting factor often is not the hardware itself. Rather, the combination of the PCI (shared bus) architecture and traditional operating-system design (which abstract device drivers, network protocols, *etc.*, by using general-purpose APIs) place a ceiling on the maximum achievable throughput long before the hardware limits are reached (for example, the maximum achievable data-encryption rate on a 4.22 Gbps 64-bit/66Mhz PCI bus is only 840 Mbps [13]). As was discussed in [17], one way to improve performance is to farm-out encryption to a dedicated node (akin to a firewall). Even in that case, however, storage capacity and network bandwidth are likely to increase faster than cryptographic processing capacity (which, in general, is limited by the same constrains as general-purpose CPUs).

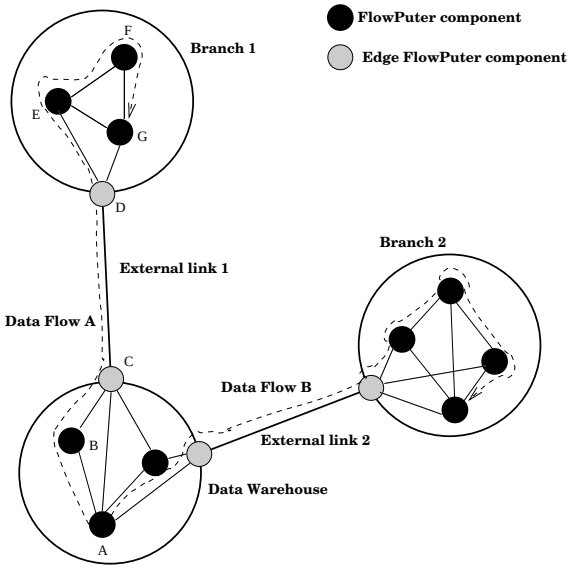

Fig. 2. **FlowPuter infrastructure distributed across 3 branches of a company.**

Consider the network shown in Figure 2. Within the perimeter of each branch, data can remain unprotected; when traversing the external links, data must be appropriately protected. To minimize the amount of records that must be encrypted, we can push part of the data-filtering predicate for Data Flow A inside the warehouse perimeter. The query optimizers of traditional databases performs a very similar task in selecting the order in which a query's sub-expressions are evaluated. So, in this example, nodes A, B, and C would be configured such that they would maximally prune Data Flow A.

A complementary approach is to encrypt each table separately, as part of a separate data flow. Assuming all the flowputer components have some encryption functionality (as the IXP425 network processor does), it may be possible to configure nodes A, B, and C to encrypt 3 independent data flows that are all destined for Branch 1; similarly, decryption can be staggered among nodes D, E, F, and G: node D can decrypt the data stream (corresponding to a particular table) that node E will need, node E will do likewise for the data stream node F will need, *etc.* The same encryption/decryption staggering can be performed for one flow: if Data Flow A is composed of only one table's data, nodes A, B, and C can interleave their encryption processing to achieve the best performance. Thus, a longer path inside the Warehouse network will allow more data to be encrypted at the cost of link bandwidth and processing capacity. Other such tradeoffs exist, and they must be taken into consideration when flow scheduling is done. Such optimization strategies have not been considered in the realm of traditional database query processing, and thus pose an interesting and open problem.

B. Access Control

Mechanisms for determining access rights to tables and records in the database (access control) have been the focus of considerable research [21], [10], [16], [7], [2], [4]. Beyond the simplistic and inadequate binary access control model (*i.e.*, properly authenticated users have access to all the records), modern databases offer considerable flexibility in defining security (access control) policies: administrators may specify user- and group-specific access control lists (ACLs) that control which tables users have access to, and of what type (*e.g.*, read-only, read-update, *etc.*) [3]. One common implementation of this is by defining *views*, and then specifying GRANT and REVOKE privileges on these views for each user or group [20]. A view is formed by joining and selecting specific columns from different tables, based on some predicate, usually defined via an SQL predicate. For example, the following SQL code snippet defines a new view, *small_transactions_anonymized*, that contains data from two other tables:

```
CREATE VIEW \
      transactions_anonymized(city,amount) AS
SELECT branch.city, transaction.amount
FROM branch, transaction
WHERE branch.city == transaction.city AND
      transaction.amount < 1000;
```

If the administrator allows a user to access only this view, then that user can execute queries using the columns present in the view (city and amount), but cannot access any other columns in tables *branch* and *transaction.* If the table is actually computed and the results stored in the database, it is called a *materialized view.* Alternatively, *virtual views* can be formed at the time a user query on that view is evaluated.

[1]For the remainder of this discussion, we only use the term encryption; integrity-protection transforms are implied.

Virtual views can be considered a form of **filter,** defining which database records the user's queries are allowed to operate on, in a manner similar to "predicate routing" [19]. It is this observation that makes this approach particularly attractive for use in our flowputer: rather than try to determine each user's access rights at the time the query is submitted, we can simply rewrite the query to operate in a virtual view defined by the user's privileges. The virtual view definition and the user's query can be combined by the (trusted) flowputer controller into a larger, *interweaved* query, which is then passed on to the query optimizer and translator. The interweaved query is then broken up into sub-queries and filters that are distributed among the appropriate flowputer processing units.

VI. Programming Languages and Compilers for FlowPuters

The architecture and application space of flowputers provides a rich research area for exploring new algorithms, programming languages, and compilation techniques to facilitate software development for flow-computing applications. Processing in the flowputer is done in the data-plane and involves classifying and manipulating nested tags at multi-gigabit per second rates across several layers of the protocol stack.

This type of classify-act-forward processing is ideal for the development and refinement of pattern-action oriented scripting languages such as Netscript or Awk for the rapid and robust development of flowputer applications. A critical aspect in the development of flowputer software and languages is the development of compilers that can translate the tag-processing functions into ultra-efficient code for the computational elements on the flowputer blades.

There are several challenges to the development of these kinds of high-performance compilers. One is the creation of efficient pattern-matching algorithms to locate the relevant fields of the nested tags. We anticipate that the classification, processing and routing of the flows needs to be accomplished in one or two processor cycles. The second and perhaps most difficult challenge is the development of code generation and optimization techniques for the network processors used in our system architecture. The use of network processors has the potential advantage of very high-performance stream processing, but network processors are notoriously difficult to program.

For example, current code generation and register allocation techniques cannot handle the idiosyncratic register banks on the IXP1200 (two general-purpose banks, two SRAM-memory interface banks, and two SDRAM-memory interfaces banks) with the quirky data-path arrangements between the banks. Transactions to memory must be performed in sets of adjacent registers - a problem that is well-known to increase the difficulty of optimal code generation even in simple machine architectures.

VII. Performance Analysis

The performance analysis of the flowputer unveils several intriguing challenges. Traditional analysis of computer systems has employed queuing theory. While queuing theory [15] has worked admirably for resource contention systems, the flowputer has certain traits that necessitate new paradigms for performance analysis. On the one hand, the constant stream of network data to process suggests that the non-linearity at 0, the bane of traditional queuing analysis, can be ignored, making life simpler. On the other hand, the typical assumption of Markovian (memoryless) arrivals used in queuing theory, deviates from the arrival pattern we expect to see in the flowputer blades. As the example in Section II suggests, different blades of the flowputer would perform different subtasks based on the installed filters. As the stream progresses between blades, the computational tasks become more and more specific and "easier" in some sense.

Here, the tools of Information Theory and Signal Processing seem appropriate for performance analysis. Each blade can be modeled as having a certain cycle budget, denoting its bandwidth. The data-flow stream has a cycle budget for each arriving packet. Carrying the analogy further, we can associate a "power spectral density" of the data stream in terms of the CPU cycles required per classify/act/forward operation — note that this also depends on the nature of the filter installed at a particular blade. The blade is then modeled as a bandpass filter. As the filter/classify/forward actions become more specific, the "bandwidth" of the flow becomes increasingly narrower. In that sense, we should be able to identify the computational capacity of a flowputer system, much like the Shannon capacity of a communication channel. The problem of scheduling data in a flowputer environment is then mapped to the problem of designing efficient codes to achieve the Shannon capacity of a communication channel[2].

This new approach to performance analysis will also lead to a better understanding of the connection between computational complexity, networking, and information theory [9], a question that has been the pursuit of researchers for quite some time. This is elucidated by a phenomenon observed last year. The spread of Code Red and Nimda viruses led to a major disruption of the Internet routing infrastructure [18]. One of the conjectured causes was that the infected machines were scanning random IP addresses on the Internet for attacks, and the edge routers next to the infected machines could not keep up with classify/act/forward demands of the wildly fluctuating destination IPs of the scan probes. A router linecard is the simplest example of a flowputer. In the Information theory/Signal processing model we have for performance analysis of the flowputer, the random IP probe data stream corresponds to a high bandwidth flow, beyond the cycle budget or the bandwidth of a linecard (blade). Simple signal theoretic analysis can then reveal the vulnerability of the linecard to data streams having a spectrum beyond its capabilities. On the other hand, flows with correlated destination networks, *e.g.*, TCP window bursts, are much easier to handle, carry a much lower cycle budget and would correspond to a low bandwidth flow.

In summary, we believe that the performance analysis of the emerging architecture of the flowputer yields both re-

[2]Note that the emphasis on throughput rather than latency makes this paradigm particularly appropriate.

search challenges as well as promising avenues for a deeper understanding of the connection between computational and communication complexity.

VIII. Conclusion

We have described a new cluster architecture that unifies switch, server and storage processing. Our architecture has significant scalability advantages over existing approaches. It offers the potential to provide wire-speed processing to applications, to give rapid access to huge increases in data capacity and to incorporate new advances in security.

The architecture provides significant advantages to application developers as well. It simplifies applications development by unifying switch, server and storage processing. It also allows application developers to take advantage of wire-speed processing, build greater resiliency and security into their applications, and exploit finer-grain parallelism than with current architectures.

References

[1] TPC-H Benchmark results. http://www.tpc.org/.

[2] E. Bertino, S. Jajodia, and P. Samarati. Supporting Multiple Access Control Policies in Database Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[3] E. Bertino, S. Jajodia, and P. Samarati. A Flexible Authorization Mechanism for Relational Data Management Systems. *ACM Transactions on Information Systems*, 17(2):101–140, April 1999.

[4] E. Bertino, P. Samarati, and S. Jajodia. An Extended Authorization Model for Relational Databases. *IEEE Transactions on Knowledge Data Engineering*, 9(1):85–101, January-February 1997.

[5] T. F. Bowen, M. Cochinwala, G. Gopal, G. E. Herman, T. M. Hickey, K. C. Lee, W. H. Mansfield, and J. Raitz. Achieving throughput and functionality in a common architecture: The Datacycle experiment. In *PDIS*, page 178, 1991.

[6] T. F. Bowen, G. Gopal, G. E. Herman, T. M. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib. The Datacycle architecture. *CACM*, 35(12):71–81, 1992.

[7] S. Castano, S. Fugini, M. Martella, and P. Samarati. *Database Security*. ACM Press/Addison-Wesley Publ. Co., New York, NY, 1995.

[8] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd VLDB Conference*, pages 146–155, 1997.

[9] A. Ephremides and B. Hajek. Information theory and communication networks: An unconsummated union. *IEEETIT: IEEE Transactions on Information Theory*, 44, 1998.

[10] R. Fagin. On an Authorization Mechanism. *ACM Transactions on Database Systems*, 3(3):310–319, 1978.

[11] J. Gray. What next?: A dozen information-technology research goals. *JACM*, 50(1):41–57, 2003.

[12] G. E. Herman, G. Gopal, K. C. Lee, and A. Weinrib. The Datacycle architecture for very high throughput database systems. In *SIGMOD Conference*, pages 97–103, 1987.

[13] A. D. Keromytis, J. L. Wright, and T. de Raadt. The Design of the OpenBSD Cryptographic Framework. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[14] M. Kitsuregawa and H. Tanaka. *Database Machines and Knowledge Base Machines*, volume 43 of *Book Series: The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publisher, Boston, MA, USA, January 1988.

[15] L. Kleinrock. *Queueing Systems, Volume I: Theory*. Wiley-Interscience, 1975.

[16] T. Lunt. Access Control Policies for Database Systems. In *Database Security II: Status and Prospects*, pages 41–52. North-Holland Publishing Co., Amsterdam, The Netherlands, 1989.

[17] S. Miltchev, S. Ioannidis, and A. D. Keromytis. A Study of the Relative Costs of Network Security Protocols. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 41–48, June 2002.

[18] D. Moore, C. Shanning, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the 2nd Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

[19] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate Routing: Enabling Controlled Networking. In *Proceedings of the First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[20] P. Selinger. Authorizations and Views. In *Distributed Data Bases*. Cambridge University Press, New York, NY, 1990.

[21] B. Wade and P. Griffiths. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, pages 243–255, September 1976.