Towards IQ-Appliances: Quality-awareness in Information Virtualization

Radhika Niranjan, Ada Gavrilovska, Karsten Schwan, Priyanka Tembey
Technical Report GIT-CERCS-07-06
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, Georgia, 30332
{*radhika, ada, schwan, ptembey*}*@cc.gatech.edu*

Abstract

Our research addresses two important problems that arise in modern large-scale distributed systems: (1) the necessity to virtualize their data flows by applying actions such as filtering, format translation, coalescing or splitting, etc., and (2) the desire to separate such actions from enterprise applications' business logic, to make it easier for future service-oriented codes to interoperate in diverse and dynamic environments. This paper considers the runtimes of the 'information appliances' used for these purposes, particularly with respect to their ability to provide diverse levels of Quality of Service (QoS) in lieu of dynamic application behaviors and the consequent changes in the resource needs of their data flows. Our specific contribution is the enrichment of these runtimes with methods for QoS-awareness, thereby giving them the ability to deliver desired levels of QoS even under sudden requirement changes – IQ-appliances. For experimental evaluation, we enrich a prototype implementation of an IQ-appliance, based on the Intel IXP network processor with the additional functionality needed to guarantee QoS constraints for diverse data streams. Measurements demonstrate the feasibility and utility of the approach.

1 Supporting Information-intensive Applications

This paper addresses data interoperability and consequent performance problems present in modern large-scale distributed applications, such as the operational information systems (OISs) used by large companies in their daily operations [18, 9] or the 24/7 engines used by online information services (e.g., online ticket pricing and reservation engine [15]). Also addressed by this work are the large data flows in online high-performance scientific applications, such as distributed real-time collaboration or remote data visualization [27]. There is an increasing gap between the data rates of such information-

intensive applications and the ability of general purpose platforms to efficiently access and manipulate that data, in addition to performing the many application-specific tasks required of these processors. To deal with this issue, it has been shown useful to associate with such platforms specialized 'information appliances', which are software components running on separable, potentially custom infrastructures that carry out tasks pertaining to data interoperability, tracking data movements and evaluating them, etc., simple examples including firewalls or intrusion detection engines. By separating such actions from the basic business logic executed by applications, it then becomes possible to dynamically add new data-centric services, to meet the needs of additional data streams or to deal with new types of data, and to dynamically manage how data streams are manipulated.

Entire businesses are based on the 'appliance' model explained above, serving specific application communities, such as the medical form conversions necessary in hospital and insurance settings, or ensuring the efficient and flexible distribution of data in wide area settings [22]. In this paper, we focus on the 'network-near' class of such appliances, which are able to execute selected data processing actions 'close' to the network itself, thereby not imposing unnecessary data movement overheads and associated memory loads on the machines that run business logic. Examples of such appliances range from firewalls and NAT boxes, to appliances for deep-packet processing, such as content-based load balancers, or units which perform select crypto- or XML-related functionality. A custom appliance used by one of our industry partners in the OIS domain extracts incoming data messages and reformats them in the internal binary data representation [18]. Examples in a search engine context are an appliance performing request load balancing across the engine's many cluster servers, or one that merges generated ticket pricing information with customized DoubleClick advertising based on current user profiles [21]. In all such cases, the purpose of these appliances is essentially, to 'virtualize' some original data stream [24], by manipulating the stream and/or creating a new stream with content suitable for some application-specific purpose.

Given the examples described above, it is not surprising that many modern enterprise systems contain separate racks of custom boxes, termed 'barnacles' in [19], each of which serves some distinct purpose. Recent industry efforts aim to consolidate this 'appliance' space, for the same power/space/increased

utilization arguments that have been driving virtualization technologies for host systems or in the network domain [17], and targeting a limited set of 'deep-packet' processing functions. Contributing to this trend, the general objective of our research is to *better understand the platform capabilities required for customizable, flexible information appliances, on top of which multiple distinct data manipulation services can be simultaneously and dynamically instantiated* [13, 8, 21]. The more specific objective pursued in this paper is to *make such appliances quality-aware*, that is, to enable them to offer (1) distinct quality levels for separate data flows, in order to maintain end-to-end QoS requirements, while at the same time, (2) efficiently utilizing aggregate platform resources.

Quality awareness is difficult to implement for several reasons. First, since it is hard, if not impossible, to statically assess the dynamic resource requirements of a certain data flow and the content-based services that operate on it, due to data-dependencies in services, static specifications have to be enhanced with runtime monitoring and assessment techniques [10]. Second, the fixed up-front partitioning of the resources is not typically feasible, as it must be based on worst case predictions and will therefore, result in low overall resource utilization. Worse, it may prevent a select application-flow or service to deliver adequate performance under increased demand, although the platform may have abundance of unused resources. Third, there may be runtime changes in the parameters that most significantly impact the quality levels experienced by a certain application flow, because these may depend on the specific application-level processing being performed and the associated quality attributes of interest (e.g., throughput vs. response time).

The specific goal of the work described in this paper is to provide a suitable basis for diverse domain-specific implementations of quality-aware information appliances, which henceforth, we term IQ-appliances. Specifically, we describe *an architecture for IQ-appliances and their runtimes,* the latter providing the basic support for flexible online monitoring, resource allocation, and adaptation needed by higher level quality management methods. Such runtime functionality makes it easy (i) to dynamically change how distinct data flows are mapped to platform resources, the goal being to adapt these mappings to better meet current flow constraints with available resources, and (ii) to drive these changes with application-level quality notions. The runtime's API can be used by applications (iii) to respond to dynamic changes

in an application specific manner and (iv) to provide the runtime with information useful for further specialization.

In summary, the main contributions of this paper are the following. We propose mechanisms that can be used to provide dynamic quality of service guarantees to applications and discuss its capabilities, including methods for continuous monitoring and configuration state management. We argue that these constitute a sufficient basis for runtime methods for data stream adaptation. We describe the realization of these mechanisms in the context of a prototype implementation of an IQ-appliance developed in our research, using the IXP network processor. Measurements of the prototype demonstrate the feasibility and utility of these 'IQ' capabilities of future information appliances. Specific experimental results concern the overheads of effective stream scheduling and the ability to dynamically adjust the appliance-level resources used by certain information streams. Current results assume that different quality properties are associated with different virtual flows passing through the appliance, but this may be generalized to differentiate different sub-streams or types of data within a single stream, as well.

Remainder of paper. The remainder of this paper is organized as follows. Section 2 describes the basic functionally and runtime support needed quality-aware information appliances – IQ-appliances. Next, we describe an appliance prototype based on the Intel IXP2400 network processing platform. Section 4 presents measurement results and their analysis. A brief summary of related work, concluding remarks, and future work constitute the next two sections.

2 IQ-Appliances: Supporting Dynamic Information Flows

2.1 Basic Concepts

We define an *IQ-appliance* as an information appliance that is able to support differentiated quality levels for distinct application-level data flows, where resource requirements may differ across flows and where each such flow may be virtualized via associated processing actions. To realize such flows, the appliance must implement the following abstractions:

- An *IQ-flow* is an abstraction for a collection of packets or application-level *data items* that can be classified according to some set of predicates. For instance, an IQ-flow may be all network

traffic associated with a certain virtual machine and identified via some virtual identifier (e.g., as supported in protocols like SCTP), or it may be a collection of all application-level data items of a certain type and with a common destination address. An example of the latter is the virtualization of a stream of business events prior to sharing them with an entity external to the company, which involves extracting company-sensitive information and reformatting them into some standard format like XML.

- A *data item* in a flow may correspond to a single or some collection of application-level messages, but it must be fully contained within a single flow and within a limited time window.
- Arbitrary processing actions, *IQ-handlers*, may be associated with each IQ-flow and applied to each data item, ranging from simple data forwarding, replication, or filtering actions, to more complex format translation, parsing, or content down-sampling operations. Examples of the latter include compression in media flows [26] and element selection in remote data visualization [27].
- A flow can specify lower bounds for QoS, which requires the IQ-appliance's runtime to provide certain minimum resource levels, independent of momentary spikes in data rates or in the amounts of processing required by other flows. Our current implementation provides such minimum guarantees, but leaves it to operating systems or applications to provide higher level methods for admission control.

We note that typically, there is substantial flexibility in how IQ-appliance resources may be allocated. For instance, runtime resources beyond those required for lower bound guarantees can be shared so as to first address the quality requirements of the highest priority or most critical flows. Experiments with different strategies for prioritization appear in Section 4 below.

2.2 Hardware Assumptions

Before describing the basic architecture of an IQ-appliance and its specific runtime components evaluated in this paper, we first digress by describing in more detail some of the assumptions we make about the hardware on which IQ-appliances run.

An IQ-appliance must deal with multiple information flows and for each flow, it must execute application-

specified operations, potentially applied to each of the flow's data items. The highly parallel nature of such processing indicates the need for many different processing cores capable of performing a range of processing actions or execute entire chains of transformations, for many concurrent flows. Modern network processors [11] have such capabilities, albeit in many cases, with some limitations in terms of the processing actions they can carry out (e.g., no floating point support, limited state per operation, etc.). We dimiss the use of parallel ASIC platforms, since that would negate the commoditization argument advanced in the introduction of this paper. Instead, we assume an appliance architecture similar to those present in current multicore machines, where parallel processing cores have access to shared memory, with a memory hierarchy that ranges from faster and smaller on chip memory (e.g., for maintaining per flow state, or runtime configuration parameters), to larger and slower off-chip memory for storing flow data. Concerning processing, it is reasonable to assume that information appliances will have certain accelerators for frequently executed communication functions, such as protocol processing, hash units for operations like classification, or even cryptographic support. IQ-appliances leverage the presence of such accelerators for application-specific transformations applied to data 'in transit' through the appliance [20].

Finally, while the bulk of the platform's computational resources are designated for fast path processing of different application flows, we assume the existence of a designated control context (e.g., a hardware thread, or a dedicated control processor), which has access to all of the platform's resources and can execute management and reconfiguration functionality. This is a reasonable assumption given the now ubiquitous presence of such management processors in bladeservers, for cluster machines, and for local area compute infrastructures [23].

2.3 IQ-Appliances: Software and Hardware Architecture

Figure 1 illustrates the main runtime components of an IQ-appliance, summarized as follows:

Networking Services – encapsulates tasks related to packet receipt (Rx) and transmission (Tx) and basic protocol-level functionality, such as fragmentation and reassembly.

Classification / Pattern Matching – enables flexible matching of incoming packets into flows, where

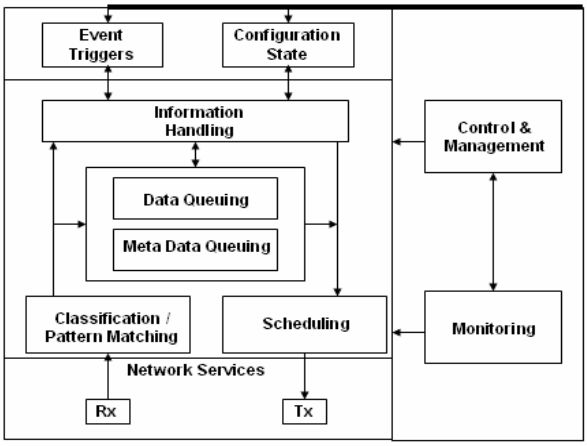

Figure 1. Runtime Components

flows may be described by their network-level parameters, such as source and destination IP addresses, virtual flow identifiers, or higher-level information contained into application-level headers, such as data type, or even application-level content.

Queuing (Buffering) – provides functionality to create and manage coordinated access to *buffers of application data* (i.e., packet payload), as well as *metadata queues* (i.e., handles for packet buffers containing header information). These shared memory buffers are the main mechanism for communication among different processing elements on the platform.

Information Handling – in order to enable the per flow execution of arbitrary application-specific functionality and to perform various transformation or filtering/forwarding actions based on flow data, the runtime provides a collection of resources (hardware or software contexts) for the execution of application provided handlers. It is the runtime's responsibility to correctly associate data events from their queues to the appropriate handler. These mappings are maintained in per flow configuration state; they can be dynamically modified.

Scheduling – a set of processing contexts (i.e., threads and/or cores) dedicated to moving processed data from the corresponding data buffers to the shared transmission resources (i.e., threads executing the Tx code and accessing the shared network ports). Similar services can be used to coordinate access to other shared resources, such as crypto units. The exact manner in which the scheduling contexts (or cycles)

are allocated to individual flows depends upon the current platform configuration state.

Monitoring – a collection of monitoring "knobs" that can be turned on or off, or whose frequency can be adjusted. These can be used to collect in a predetermined location, information about the current platform state, from individual queue lengths, to per-flow packet delays, etc.

Configuration State – a description of the current values of all of the runtime's configurable parameters, such as the fields used in the classification process, the number of buffer queues, number of cycles or threads dedicated to servicing a particular queue, monitoring behavior, etc.

Control & Management – a separate control entity, which is a software thread or a dedicated control context that is responsible for interaction with external application components and with the management infrastructure and for mapping the externally specified policies or quality requirements into concrete platform configurations. This is done in a manner that is safe with respect to the current runtime operations (i.e., to obtain currently held locks, to prevent inconsistent state of data currently processed, etc.). The actual nature of these mappings is not the focus of this paper and will be further investigated in future work. Finally, due to resource limitations on the physical appliance platform, the control and management functionality may be partially executed on an *external, general-purpose, configuration core*.

Event Triggers – the runtime can generate events associated with different states or changes in operating conditions, such as data rates, queue lengths, etc., with which application-specific response handlers can be associated. For instance, under high load conditions, a low priority flow may choose to apply its custom data handler, rather than letting the runtime drop arbitrary data packets due to buffer overflow.

As illustrated in Figure 1, once classified, data may be immediately enqueued for later processing, or it may immediately be passed for information handling. This implies that the runtime supports *pre-queuing* data handling, for instance, so as to filter out unneeded data before any loads are placed on the platform's memory subsystem. Similarly, we support *post-queuing* data handling, needed for instance to inline certain data manipulations jointly with the transmission-related operations, as in packet and/or header compression to meet throughput levels available on a lower speed link, or to facilitate link fragmentation and interleaving to prevent large packets from causing head of line blocking for delay sensitive data.

The ability to apply in-transit arbitrary application handlers has been shown useful in previous work [7, 6]. Toward this end, we represent application-specific processing codes as lightweight IQ-flow handlers, which can be efficiently applied to data at various points in the data path as it traverses the network. One of the key properties of these handlers is their ability to efficiently interpret, access, and manage the layout of application-level data and the contents of its fields across a set of network packets, i.e., a data unit. A requirement imposed on IQ-flow handlers is that the runtime exposes an API that can provide information about the behavior these application-provided codes, such as instruction counts, need for special purpose hardware, whether they are CPU or I/O bound, etc. Such *IQ-flow handler profiles* can be used to further optimize runtime behavior. We are not concerned with how this information is obtained, but there are multiple options. One option relies on the tool chain used to create handlers. Another option is to have the runtime maintain monitors and counters to extract these characteristics of its various loads. Further discussion of IQ-flow handlers appears in [2].

2.4 Dynamic Platform Resource Management

Next, we describe the set of mechanisms and accompanying data structures needed to permit the dynamic management of platform resources and their use for servicing different IQ-flows. The basis for supporting dynamic resource management lies in the following platform capabilities: (1) flexible and configurable monitoring of platform parameters, (2) rich configurationation state regarding a range of runtime properties, and (3) the ability to perform dynamic platform resource allocation.

Resource management points. Our approach to enabling dynamic platform resource allocation is to enrich the datapath with Resource Management (RM) points. These are execution points in the fast path where it is safe to perform reconfiguration operations, e.g., at data item boundaries. At RM points, individual threads/cores participating in the fast path data processing inspect the configuration state (on a per thread basis) to determine the set of actions in which they should be involved. Examples include (i) determining monitoring frequency and the platform parameters to be monitored, (ii) the IQ-handlers to be invoked for particular data items, (iii) the data queue to be serviced based on QoS agreements and scheduling policy, etc.

Configuration state. Platform-resident configuration state is used for multiple purposes. First, it can represent individual flows and the processing functionality associated with flow data, i.e., classification of incoming data messages into appropriate platform queues, and invocation of corresponding IQ-handlers. Second, it is used to allocate platform resources to individual flows. Specifically, at RM points, each processing context (i.e., thread), inspects its configuration state to determine which flow (i.e., queue) or set of flows it should service next. In order to minimize the overheads encountered during this operation, we assume that in the case of multiple flows, these are already ordered according to their priority by the external configuration core. Finally, portions of the configuration state describe the platform monitoring functionality – the concrete set of platform parameters to be monitored, and the frequency with which their values are to be inspected.

This state is organized as global, per-flow, and per-thread tables. Access to the appropriate state components is enabled either by static offsets (for static global or per-thread data structures), or via a flexible and configurable classification process. Such a classification mechanism is necessary to map variable length flow descriptors that may span network-headers (i.e., source and destination IP addresses) or application-headers (i.e., binary format descriptors) to a fixed, smaller-bit size unique identifier. The ability to provide flexible classification like this lies in the selection of parameters to the hash function polynomial, the details of which are not the focus of this paper.

Clearly, efficient access to various components of this state is of critical importance to platform performance. Due to its potential size, it is important to realize that it may not be feasible to maintain all of this state in fast, on-chip memory. Instead, we rely on distributing state components along the memory hierarchy on the target hardware platforms (see Section 2.2), and dynamically placing state components in appropriate memories. A specific example are the large classification tables all of which cannot reside in fast memory at all times. To accommodate this fact, we extend the platform-level flow identifier with location attributes that identify current table location.

Configurable monitoring. Performance information, either regarding the appliance platform overall, or regarding individual per-flow service levels, can be extracted from a range of runtime parameters, such as core CPU utilization, device Rx/Tx rates, or queue lengths. The exact values to be monitored are

described in a subset of the configuration state, along with the frequency of this operation. Monitoring is a distributed mechanism, in that each core monitors the above mentioned values separately, and updates the corresponding resource state entries accessible from the control core.

Consistent operation. Changes in configuration state may be driven by external action – pushed onto the appliance platform through the external configuration core, or as a result of runtime changes. The control core is involved in pushing the state updates to their appropriate locations in the configuration state, and through the use of efficient bitwise locking scheme, to ensure consistency with respect to fast path flow processing.

Further optimizations. Our prior work has demonstrated the feasibility and utility of dynamic code swapping [13]. While in the current approach, the runtime actions are interpreted by accessing corresponding state elements, we acknowledge the utility of dynamically hotswapping new instruction store contents, and executing customized generated code. This results in additional performance benefits due to elimination repeated memory accesses.

3 Implementation Detail

We next describe in more detail a concrete realization of an QoS-aware information appliance.

Intel IXP2xxx Network Processors. Our prototype implementation is based on the Intel IXP2400 network processors, attached to standard Linux-based hosts. As with other such platforms, the key attribute of the Intel 2400 is the presence of multiple on-board processing engines, termed microengines, which can be organized in arbitrary manner to perform certain tasks. Important features of the IXP2400 architecture include: (1) specialized hardware support for network operations, (2) eight 8-way multithreaded microengines for data movement and processing, (3) a Linux-based Xscale core for management and other functions, (4) a rich memory hierarchy including fast on-chip local (i.e., per microengine) and global scratchpad memory, as well as 128MB of off-chip SRAM and 256MB of DRAM. For more detailed information on the IXP2400, we refer the reader to [12].

IXP2400 as a QoS-aware Information Appliance. The realization of the appliance prototype and the accompanying resource management mechanisms described in Section 2 can be summarized as follows.

The processing cores, i.e., the execution of fast path tasks like data receipt and transmission, and the application-provided IQ-handlers, are mapped to threads on the IXP's microengines. The IXP appliance is attached to an x86 host, which serves as the external configuration core. The control core functionality is executed by threads on a single dedicated microengine, with an XScale-resident component involved in signalling and communication with the external processor. RM points are embedded into fast path operations, and they are invoked with configurable frequencies, i.e., after processing each n appliacation-level data items. Monitoring operations are also executed at RM points, potentially with different granularity, and currently focused on monitoring various data rate and queue length indicators.

An IQ-flow is identified as a combination of networking information, i.e., source address, and application-level information comprised of an efficient binary data type descriptor [4]. The classification process maps this into a configurable smaller bit-size identifier and classifies incoming data into DRAM-resident data buffers. Metadata queues are maintained in faster SRAM memory, with queue desciptors (including tail and head pointer, QoS-related information, etc.) residing in on-chip Scratchpad memory. The configuration state is distributed across microengines' local memories, Scratchpad and SRAM.

Current resource management operations focus on allocating microengine threads to service distinct IQ-flows. Such thread scheduling is described in per thread information, updated by the control core. It can result in several thread allocation schedules, further described and evaluated in Section 4.

4 Experimental Results

Experimental Setup. We next discuss the results from the experimental analysis of the prototype IQ-appliance. The objective is to demonstrate that adaptive runtime resource allocation can result in a system with better overall utilization of shared resources. Low adaptation overheads make it possible to rapidly adapt to changes in runtime parameters and workload requirements, while still maintaining acceptable QoS levels for each of the distinct flows being handled. The experiments are conducted on a cluster of 2850 Xeon nodes, each with an IXP2400 network processor card, interconnected via 1Gbps Ethernet. One of the IXP cards is used as the network appliance being evaluated, while other nodes are used as data sources and destinations. Since the hardware platform used in this prototype implementation

Round Robin (**RR**)	each thread polls the different flows in a round robin manner
Equal Static Allocation (**EAS**)	threads are statically assigned to flows in a uniform manner
Static Allocation (**SA**)	threads are statically allocated to flows according to priority levels
Priority Aware (**PA**)	threads pull packets in the order of priority; lower priority flows are serviced only when no higher priority packets exist
Priority Aware with Reservation (**PAR**)	a number of threads is preallocated to flows according to their priority (as in SA), while the remainder are shared in a priority-aware manner (as in PA)

Table 1. Resource Allocation Schemes

is capable of handling much more than the 1Gbps data rates which we can deliver to it in our testbed, we artificially limit the capacities of the outgoing links to 500Mbps.

In the following experiments, 3 flows A, B, C suffice for demonstrating the notion of QoS 'built into' the appliance. The flows are assignd different amounts of runtime resources. For instance, flow A has a metadata buffer of 512 words which is twice the size of B's and C's metadata queues. Each application flow also has some corresponding processing, which will be explained as and when necessary. The problem tackled below is to allocate computational resources to the different flows in the system, to ensure QoS guarantees at all points of time. This could also be viewed as a control system problem, where some feedback control is required to keep the system stable. In order to evaluate the QoS levels delivered to each flow, we vary the way in which the runtime allocates available resources, i.e., threads, to each flow, according to the resource allocation schemes described in Table 1. In the experiments described in the remainder of this section, flow B is always assigned the highest priority.

Importance of Runtime Quality-Awareness

The case for priority-awareness. The first set of experiments evaluates the different policies for resource (i.e., threads) allocation to flows, where each flow consumes equal amounts of the ingress bandwidth (i.e., data items from each flow arrive at approx. 330Mbps). The throughput measurements in Figure 2 show that while priority-sensitive schemes exhibit certain performance gains, the overall improvement in throughput levels is on the order of 10% for the high priority flow, and still almost 50% less than its incoming rate. Latency improvements, are however more pronounced, as can be seen in Figure 2 since data is dispatched more promptly form higher priority queues. These results demonstrate (1) the

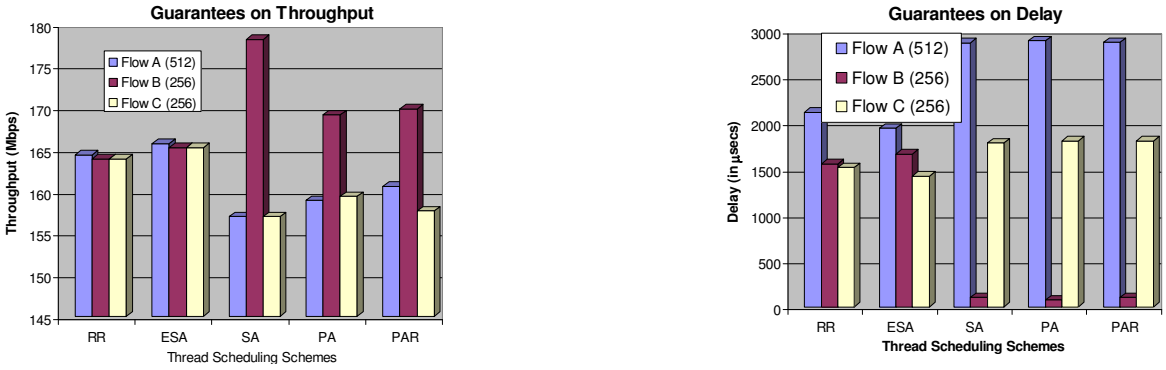

Figure 2. Throughtput and Delay with different thread allocation policies

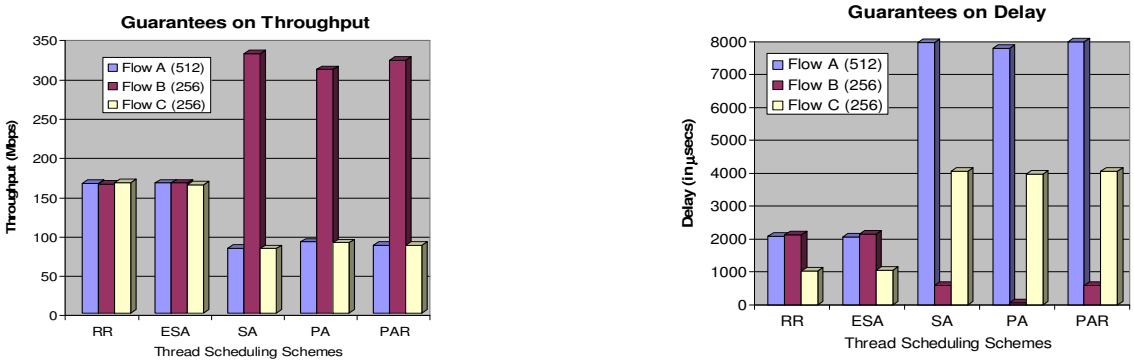

Figure 3. Throughput and delay with pre-queueing/ingress IQ-flow handling

importance of enabling some form of quality-awareness in the runtime, and (2) the need for greater coordination at multiple levels throughout the data path through the system.

The need for pre-queuing IQ-flow handling. Next, we exploit the runtime ability to associate application specific behaviors (i.e., codes) with select changes in the runtime operating conditions. Such event triggers can be exploited to permit graceful degradation of the service level under overloaded conditions. For instance, we may want to apply application-specific filtering actions to the less important flows (e.g., drop data in A and C if the respective queues are full), in the event of a critical event occurrence in the high priority flow (i.e., B). Such filtering actions are most suitable for pre-queuing execution, and are associated with the Rx codes. The results in Figure **??** illustrate the throughput and latency levels observed under different platform configurations when such Rx-side execution of application-level handlers/codes is permitted. We refer to this as Priority-Aware Rx. We observe that throughput levels for the high priority queue finally reach the desired input levels. The graphs also demonstrate

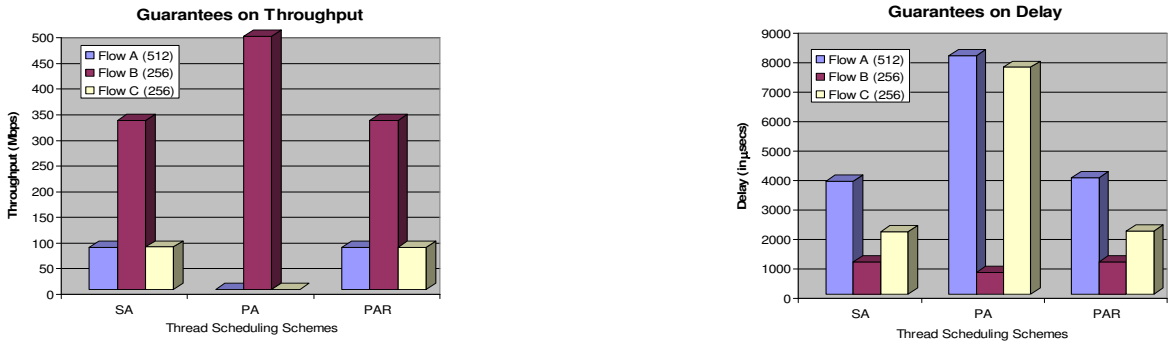

Figure 4. Importance of minimal resource reservation.

that providing platform events to which the application can respond in some custom manner, without having additional platform support to enforce different quality levels is insufficient (see bars for Priority Unaware RR or SA in both latency and bandwidth graphs.).

Importance of resource reservation. In order to guarantee certain minimal quality-levels, which are particularly important in virtualized environments, especially in the presence of dynamic variations in the workloads, it is necessary to rely on resource reservation. The results in Figures 4 are gathered for unequal input rates, with flow B's rate reaching 700Mbps, and flows A and C consuming the remaining 300Mbps on incoming bandwidth. The results show that under spikes in the higher priority workload, a purely priority-driven runtime behavior may be optimal for the high priority flow/service, but it may result in unacceptably low service levels for the remaining flows.

We present the results only for Priority Aware(PA) and Priority Aware with Reservations (PAR) schemes. PA completely starves the lower priority flows in order to service the higher priority flows, and drastically increases the observed delays. Such purely priority based resource allocation policy is therefore unacceptable in the aforementioned environments.

Thus, we summarize the results as follows: (1) priority-aware resource allocation schemes, such as SA, PA and PAR, are important for delivering differentiated service levels to incoming data flows; (2) associating IQ-handlers with platform event triggers permits applications to dynamically install pre-queuing IQ-handlers, which further improve the platforms' ability to meet application-level quality requirements; and (3) coupling priority-aware resource allocations with mechanisms for per-flow minimal

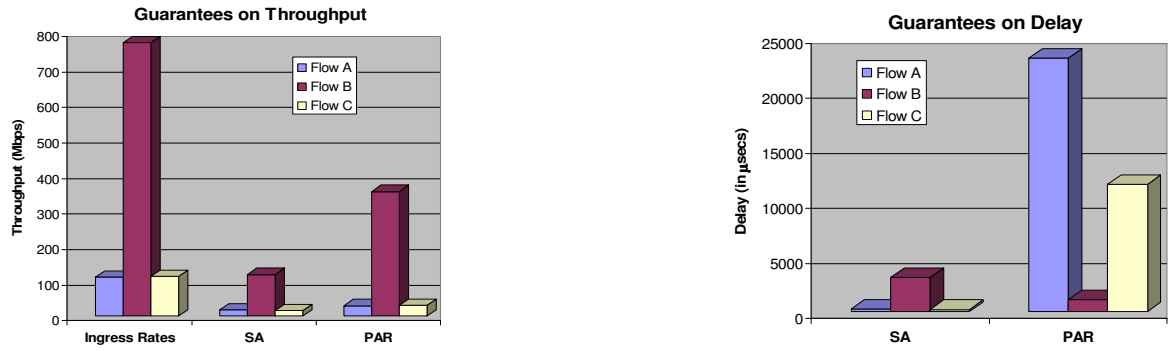

Figure 5. Importance of dynamic resource allocation

resource reservations is necessary to avoid unacceptable service degradation. These observations cause us to focus the remaining experimentation on the SA and PAR schemes, as they are the only ones which provide for priority-based resource allocation.

Improved utilization of platform resources. Focussing on the tradeoffs between dedicated resource usage and limites in the availability of shared resources, we see that the comparison of SA and PAR in Figure 5 indicates that with SA, the outgoing link reaches barely 40% utilization. In the second case, PAR ensures sharing of runtime resources, so that the outgoing link is fully utilized (500Mbps cumulative throughput). Interestingly, even the throughput levels for the lower priority flows experience a slight increase. Overall, with the PAR scheme, we attain more than a 300% improvement in throughput levels and more than a 50% latency reduction to the critical flow.

Feasibility of achieving quality-awareness.

Monitoring overheads. The next set of experiments aims to understand the overheads and impact of continuous platform monitoring. The resources being monitored are specified as part of the runtime's configuration state, and as their size and type vary, so do the overheads. For simplicity, we distinguish between 'Simple' and 'Fancy' monitoring, which differ significantly in the number of runtime parameters being monitored. Simple monitoring tracks the aggregate resource utilization at a particular priority (i.e., QoS) level, whereas Fancy monitoring further differentiates the resources consumed by individual flows within a QoS level. In addition, we also support Queue Length Monitoring, which reports only the lengths of the different queues in the system. The results in Figure 6 indicate that even with Fancy

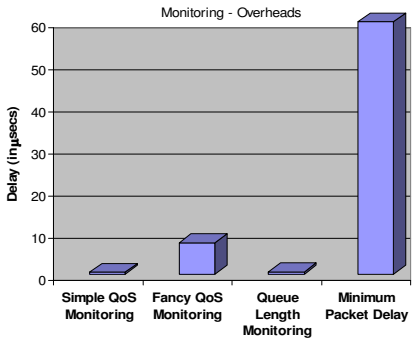

Figure 6. Monitoring Overheads

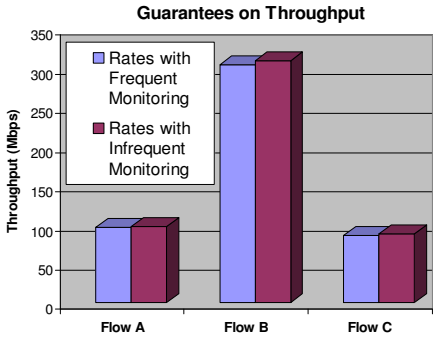

Figure 7. Monitoring Effects on Rates

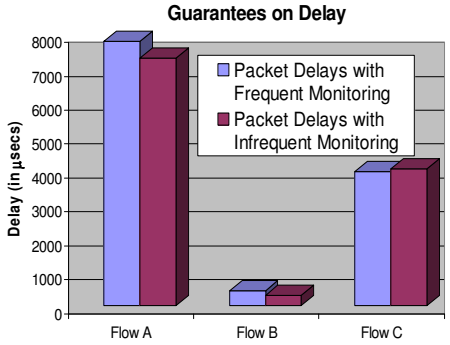

Figure 8. Monitoring Effects on Delay

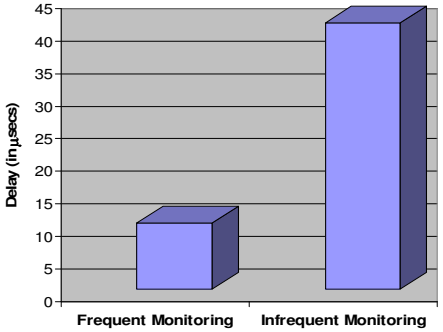

Figure 9. Monitoring Response Time

Monitoring, when we monitor all runtime parameters, in the worst case, we observe a latency increase of at most 12.4% compared to the minimum packet delay for the highest priority flow. The overheads are significantly lower with respect to other flows and data sizes in the system.

Next, we evaluate the impact of varying monitoring frequency. For frequent monitoring, the priority specifications are checked at every context, i.e., distributed monitoring. For infrequent monitoring, a single control context is given the responsibility of reading the priority specification whenever it gets a chance to run. Figure 7 shows that the frequency impact on packet throughput is negligible and remains within few percent for different flow priorities. More interestingly, due to the parallel nature of the execution platform, and the hardware supported thread scheduling, which performs context switches only when threads perform I/O operations, increasing the monitoring frequency may even reduce message delays. The results in Figure 8 show negligible delay increases for one of the three flows (C) and more

visible delay decreases for the other two flows. Finally, we also evaluate the runtime responsiveness to changes in some of the monitored parameters, as a function of the monitoring frequency. We observe that even with infrequent monitoring, we can detect changes in runtime parameters within $45usec$, which is less than 2/3 of the minimum packet delay. This is again due to the fact that the monitoring functionality is distributed across multiple hardware supported contexts in the parallel platform.

Classification overheads. We also evaluate the overheads associated with other runtime mechanisms described in Section 2. The results in Figure 10 show that accesses to configuration state distributed across DRAM, SRAM and Scratch can be executed with negligible overheads, of less than $0.3usec$ for state in DRAM, including the classification operation. Updates to individual state entries can be performed with maximum $1usec$ penalty (again, for DRAM-resident state), and the maximum observed delay to consistently propagate updates from the external configuration host to individual fast path processing thread is less than $100usec$.

Dynamic adaptation.

A set of experiments validates the platform's ability to dynamically adapt to changes in operating conditions. Figure 11 demonstrates the adaptability of the system. The bars marked Input track the ingress flow rates over time, and the interleaved bar sets marked with Dynamic Adaptation Scheme (DAS) report the observed egress rates. Initially, the resource allocation policy gives equal priority to all flows as long as it can keep up with their rates (as shown in the first pair of bars). As soon as the control core detects the ingress rates increases at time *t2*, it switches to a resource allocation policy of priority scheduling with reservations, so as to give maximum resources to the critical flow B, while still meeting the minimal QoS levels for the remaining flows (see the second pair of bars). As a result, flow B's data is processed at line rates, while the remaining flows receive minimal bandwith levels. Furthermore, when B's ingress rate is reduced at *t3*, this policy results in appropriate adjustments so that the remainder of the available resources is allocated to flows A and C. Note that at all times, the maximum available outgoing bandwidth of 500Mbps is fully utilized. Similar behavior can be observed if multiple data streams are classified in flows.

Multi-instance appliances.

Operation	Overhead
Classify/DRAM	0.256 µsec
Classify/SRAM	0.1 µsec
Classify/Scratch	0.095 µsec
Update single entry	1 µsec
Maximum consistency delay for updates from external host	99.86 µsec

Figure 10. Runtime overheads

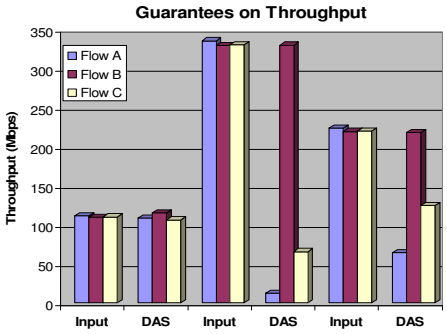

Figure 11. Adaptive Scheduling

Our final results results use replays of actual enterprise data, collected at one of our industry partners, Delta Air Lines. Here, three different processing actions, i.e., IQ-flow handlers, are applied to select data items, depending on their type. Consider data items from multiple sources at an airport that need to be exchanged with various system components. Flow A consists of updates to an external service, such as caterers. Delta data needs to be purged from sensitive information and its format standardized, resulting in 100B data item. Flow B consists of airport display updates. The processing and format translation actions for this data type are simpler, but of higher priority, and result in a 300B data item. Finally, flow C consists of state updates for fault-tolerance reasons, which result in a limited number of data accesses and minor size changes. The incoming data streams consist of a uniform distribution of all data types.

The results depicted in Figure 12 lead to two conclusions. First, they demonstrate the feasibility of supporting a mix of data handlers on a single 'multi-appliance' platform. Second, they ilustrate that even under different processing requirements for individual flows, the proposed appliance design, where platform-level parameters are monitored and used to drive the selection of 'scheduling' actions, result in most desirable performance levels. We observe that the most important flow receives the highest percentage of available outgoing bandwidth, with lowest latencies. The remaining bandwidth is evenly distributed amongst updates from the other flows, with flow A experiencing larger per-data item delays due to the significantly greater complexity of the data handler itself. Additional experimentation (not reported here) with data streams consisting of larger data sizes, based on the Delta data and with similar handler behaviors, demonstrate that data sizes do not affect the platforms ability to continue to

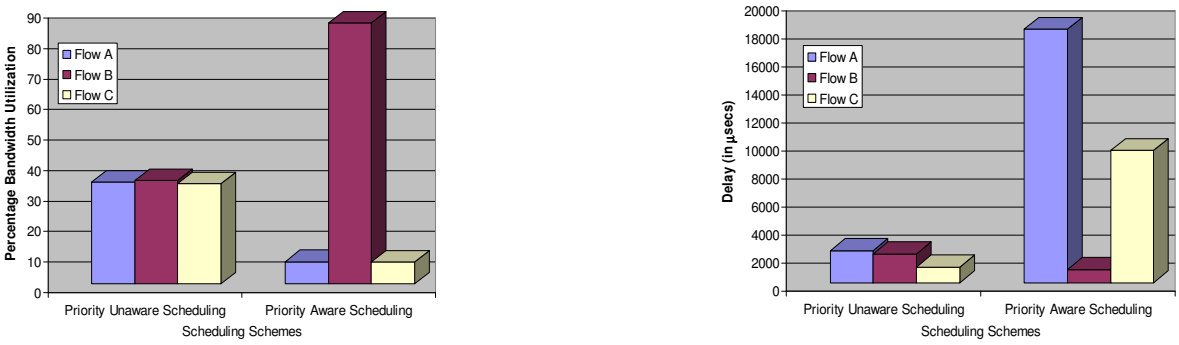

Figure 12. Bandwidth utilization and delay for Delta Airlines data with three different IQ-handlers.

demonstrate the same desired behavior.

5 Related Work

The work presented in this paper builds on our previous research to enable dynamically extensible communications platforms capable of efficiently running a wide range of application-level processing functions, represented via stream handlers [7, 13]. Such in-transit data manipulation services have been addressed by a large body or work, from active networking research [25], to research on augmenting the functionality of communications devices for select application tasks [20, 16]. Research on benefits of offloading some procssing to Network Processors is also presented in [28, 3]. This paper differs from this and our own prior work, first by exploring in detail the runtime capabilities needed for efficiently utilizing aggregate platform resources, while still meeting QoS requirements for individual application flows. Second, we specifically address the runtime functionality needed for customizing information appliances, and permitting their concurrent use for multiple types of application processing, across multiple flows. The ability to modify the original data stream, and personalize it, or according to a set of criteria, customize it for its final destination, has been previously termed data or information virtualization [24]. This paper uses this term to denote application-specific data handling.

Finally, the runtime capabilities advocated in this paper rely on continuous platform monitoring and adaptation. Similar techniques have been widely used for tuning and monitoring distributed enterprise systems [1, 2], to deploying codes in wide area infrastructures [14], to adaptive QoS for delivering

scalable media to end systems [5], etc.

6 Conclusions and Future Work

This paper describes a runtime environment for the important class of 'information appliances' typically found in modern distributed infrastructures. The approach taken (1) enables the creation of extensible appliances that can be customized to execute diverse application-specific processing actions, (2) enriches the runtime with QoS-aware functionality, IQ-appliances, so that the specific quality requirements of individual data flows can be met, and (3) includes continuous monitoring and adaptation capabilities so as to optimize platform utilization and attain individual service levels, while still honoring flows' minimum quality guarantees. We argue that the use of such appliances is particularly important for the enterprise systems and applications, where increasing levels of system virtualization are creating an ever-increasing number of information flows with different needs and of different types.

In ongoing and future work, we expect to be able to repeat or further enhance the experimental results with data traces and behaviors that match real-life conditions gathered at some of our partner enterprise institutions. One avenue of future work will integrate these runtime capabilities with a virtualized network interface device developed by our group, so as to further enhance our ability to create end-to-end virtualized distributed platforms. We will also focus on better understanding the various impacts of mixing shared and dedicated platform resources to our ability to attain desired services levels across a mix of workloads and application and VM behaviors.

References

[1] S. Agarawalla, G. Eisenhauer, and K. Schwan. Lightweight Morphing Support for Evolving Middleware Data Exchanges in Distributed Applications. In *Proc. of International Conference on Distributed Computing Systems*, 2005.

[2] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan. QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems. In *The 3rd IEEE International Conference on Autonomic Computing*, 2006.

[3] C. Albrecht, J. Foag, R. Koch, and E. Maehle. DynaCORE-A Dynamically Reconfigurable Coprocessor Architecture for Network Processors. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2006.

[4] F. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient Wire Formats for High Performance Computing. In *Proc. of Supercomputing 2000*, Dallas, TX, Nov. 2000.

[5] A. Campbell and G. Coulson. QoS Adaptive Transports: Delivering Scalable Media to the Desk Top. In *IEEE Network Magazine*, 1997.

[6] A. Gavrilovska, S. Kumar, S. Sundaragopalan, and K. Schwan. Platform Overlays: Enabling In-Network Stream Processing in Large-scale Distributed Applications. In *15th Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05)*, Skamania, WA, 2005.

[7] A. Gavrilovska, K. Mackenzie, K. Schwan, and A. McDonald. Stream Handlers: Application-specific Message Services on Attached Network Processors. In *Proc. of Hot Interconnects 10*, Stanford, CA, Aug. 2002.

[8] A. Gavrilovska, K. Schwan, and S. Kumar. The Execution of Event-Action Rules on Progammable Network Processors. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS 2004), held with ASPLOS-XI*, Boston, MA, 2004.

[9] A. Gavrilovska, K. Schwan, and V. Oleson. Practical Approach for Zero Downtime in an Operational Information System. In *Proc. of 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

[10] C. D. Gill, F. Kuhns, D. Levine, D. C. Schmidt, B. S. Doerr, , and R. E. Schantz. Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems. In *Proceedings of the 1st International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, Phoenix, Arizona, Nov. 1999.

[11] Intel Corporation. *Intel Network Processor Family*. http://developer.intel.com/design/network/producs/-npfamily/.

[12] Intel Corporation. *Intel IXP2400 Network Processor: Flexible, High-Performance Solution for Access and Edge Applications. White Paper*, 2003.

[13] S. Kumar, A. Gavrilovska, K. Schwan, and S. Sundaragopalan. C-Core: Using Communication Cores for High Performance Network Services. In *Proc. of 4th Int'l Conf. on Network Computing and Applications (IEEE NCA05)*, Cambridge, MA, 2005.

[14] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-Aware Distributed Stream Management using Dynamic Overlays. In *Proc. of 25th IEEE International Conference on Distributed Computing Systems (ICDCS-2005)*, Columbus, OH, 2005.

[15] V. Kumar, M. Mansour, B. Martin, J. Mehalingham, and K. Schwan. Policies for Enterprise Information Systems: Two Case Studies. In *10th IFIP/IEEE Symposium on Integrated Management*, Munich, Germany, May 2007.

[16] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Int'l Conference on Supercomputing (ICS '03)*, 2003.

[17] Network content flow analysis and virtual machine technologies solutions from netronome. www.netronome.com.

[18] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin. Operational Information Systems - An Example from the Airline Industry. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Oct. 2000.

[19] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. In *Proc. of HotNets-III*, Cambridge, MA, Nov. 2004.

[20] M.-C. Rosu, K. Schwan, and R. Fujimoto. Supporting Parallel Applications on Clusters of Workstations: The Virtual Communication Machine-based Architecture. *Cluster Computing, Special Issue on High Performance Distributed Computing*, May 1998.

[21] P. Royal, M. Halpin, A. Gavrilovska, and K. Schwan. Utilizing Network Processors in Distributed Enterprise Environments. In *5th Int'l Conf. on Network Computing and Applications*, Cambridge, MA, 2006.

[22] Tibco Software Inc. *Tibco ActiveEnterprise: XML Tools*. http://www.tibco.com/solutions/products/-extensibility/.

[23] Server virtualization and virtualization infrastructure management solutions from virtualiron. www.virtualiron.com.

[24] L. Weng, G. Agrawal, U. Catalyurek, T. Kurc, S. Narayanan, and J. Saltz. An Approach for Automatic Data Virtualization. In *HPDC*, 2004.

[25] D. J. Wetherall. Active Network Vision and Reality: Lessons from a Capsule-based System. In *Proc. of he 17th ACM Symposium on Operating System Principles (SOSP'99)*, Kiawah Island, SC, Dec. 1999.

[26] Y. Wiseman, K. Schwan, and P. Widener. Efficient End-to-End Data Exchange Using Configurable Compression. In *Proc. 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, 2004.

[27] M. Wolf, H. Abbasi, B. Collins, D. Spain, and K. Schwan. Service Augmentation for High End Interactive Data Services. In *Proceedings of Cluster 2005*, 2005.

[28] L. Zhao, Y. Luo, L. N. Bhuyan, and R. Iyer. A network processor-based Content-aware switch. In *Micro*, 2006.