Unification of Publish/Subscribe Systems and Stream Databases
The Impact on Complex Event Processing

Joe Sventek and Alex Koliousis
University of Glasgow

What's the problem?

- An increasing demand for complex event processing of ever-expanding volumes of data in an every-growing number of application domains
- Many of these scenarios require the ability to retain local state between constituent events of a complex pattern **and** access to relevant persistent state
- The sheer volume of data to be processed demands that we revisit monitoring architectures, especially in extremely time-sensitive domains – i.e. performance is of increasing importance

University of Glasgow

What's current practice?

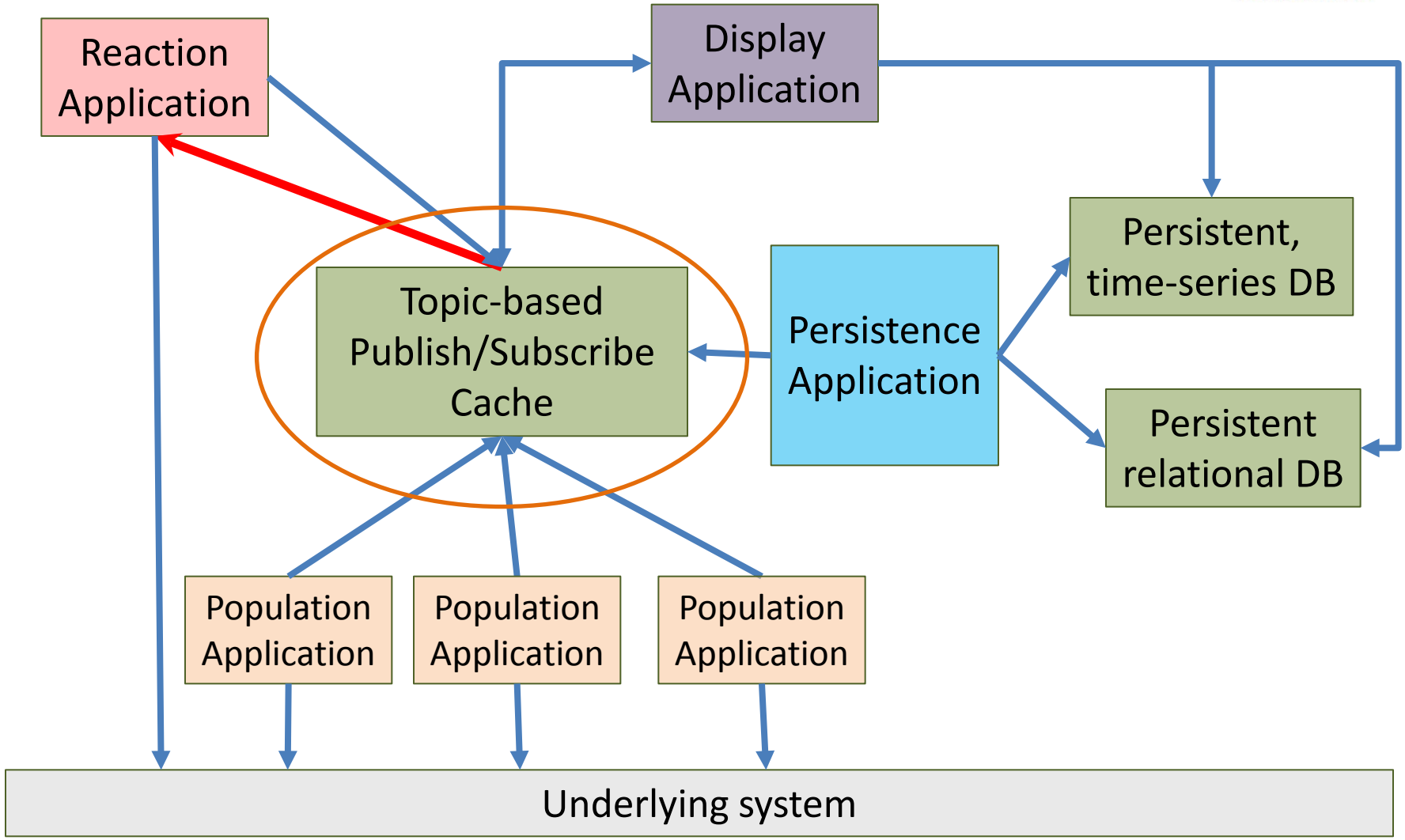
HOMEWORK

- Stream database management systems
  - with user-defined aggregate functions (e.g. Aurora)
  - with non-deterministic finite automata (e.g. Cayuga)
  - with support for multiplexing/demultiplexing packet streams (e.g. Tribeca)
  - with two-level query architecture to push logic closer to high rate sources of data (e.g. Gigascope)
  - others that dispense with SQL altogether

The Homework Project

- Create a home network router that passively monitors all traffic in the home network, replacing the commodity router
- Make this monitored data available in real-time to display, persistence, and reaction applications
- Provide mechanisms for detecting complex event patterns in the monitored data and that can trigger management policies
- Provide display and control functionality to home users that is intuitive to navigate and use
- Iterative design, implementation, and deployment strategy

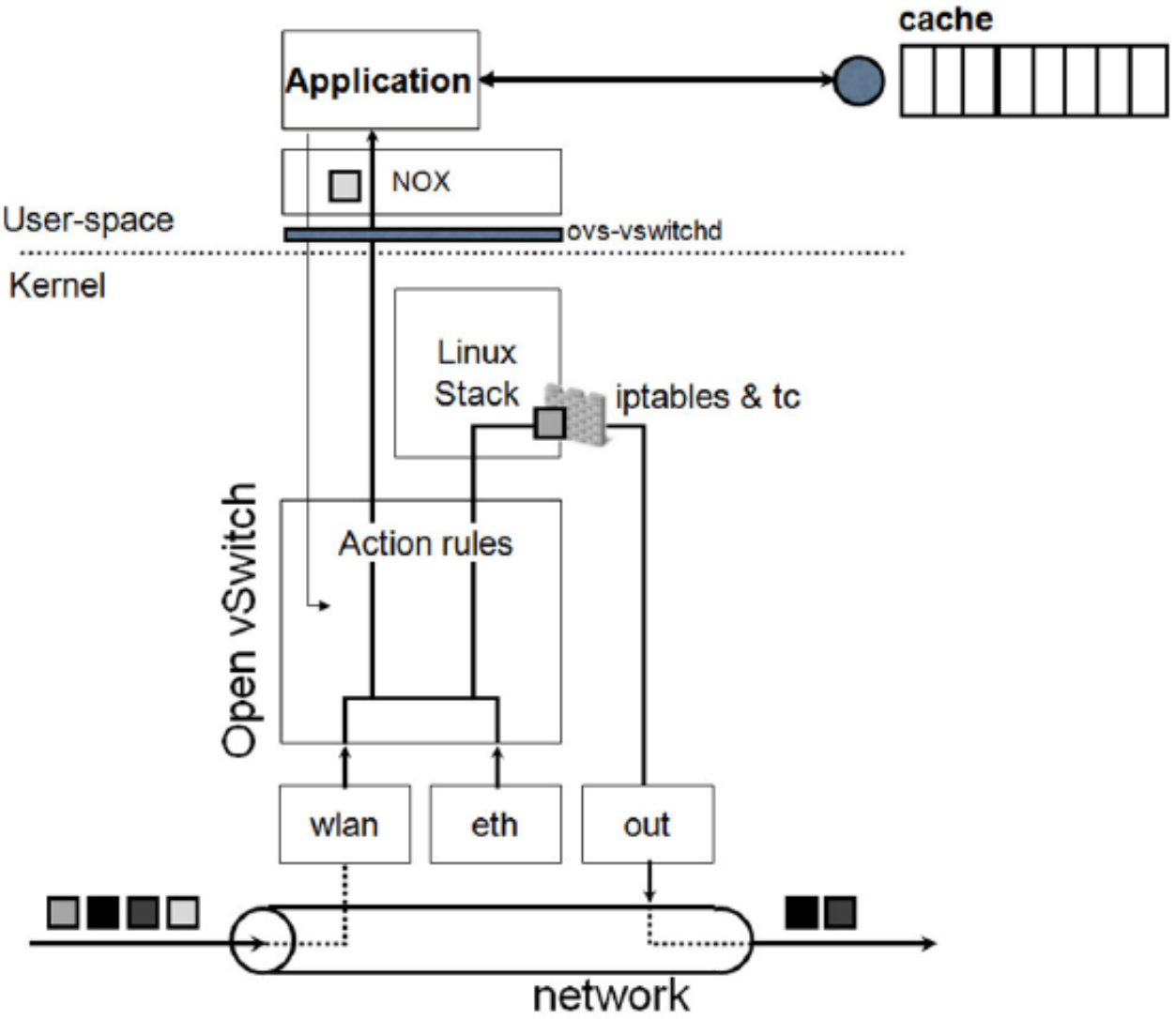
Information Plane Architecture



Why something new?

- Initial Homework experience pointed to several needs that were difficult to meet with any particular stream database system
  - Need for rapid access to persistent data as part of complex event pattern matching ➔ support persistent relations in the Cache
  - Need for periodic. *ad hoc*, access to most recently received events to drive user interfaces ➔ provide CQL interface to event data streams
  - Need to support programming of logic to detect complex events in real-time based upon both local and global state ➔ GAPL
  - Requirement for maximal flexibility in architecting the monitoring system

Underlying Homework system



University of Glasgow

The Cache

HOMEWORK

- **Topic-based publish/subscribe cache**
  - pub/sub topics defined as stream database tables
    - ephemeral tables in which the primary key is time of insertion
    - persistent tables in which the primary key is the first defined field
    - tuples in all ephemeral tables are stored in a circular memory buffer
    - tuples in persistent tables are stored in the heap
  - Ephemeral – continuous, potentially large volume of measurements $\Rightarrow$ the Cache cannot possibly make it persistent, so don't even try …
  - Real-time $\Rightarrow$ must optimize use of resources to keep up with the measurements
  - Ordering of tuples in tables is by time of insertion
  - Supports stream database view of time-series data
  - Innovative approach – "raw" events are aggregated measures

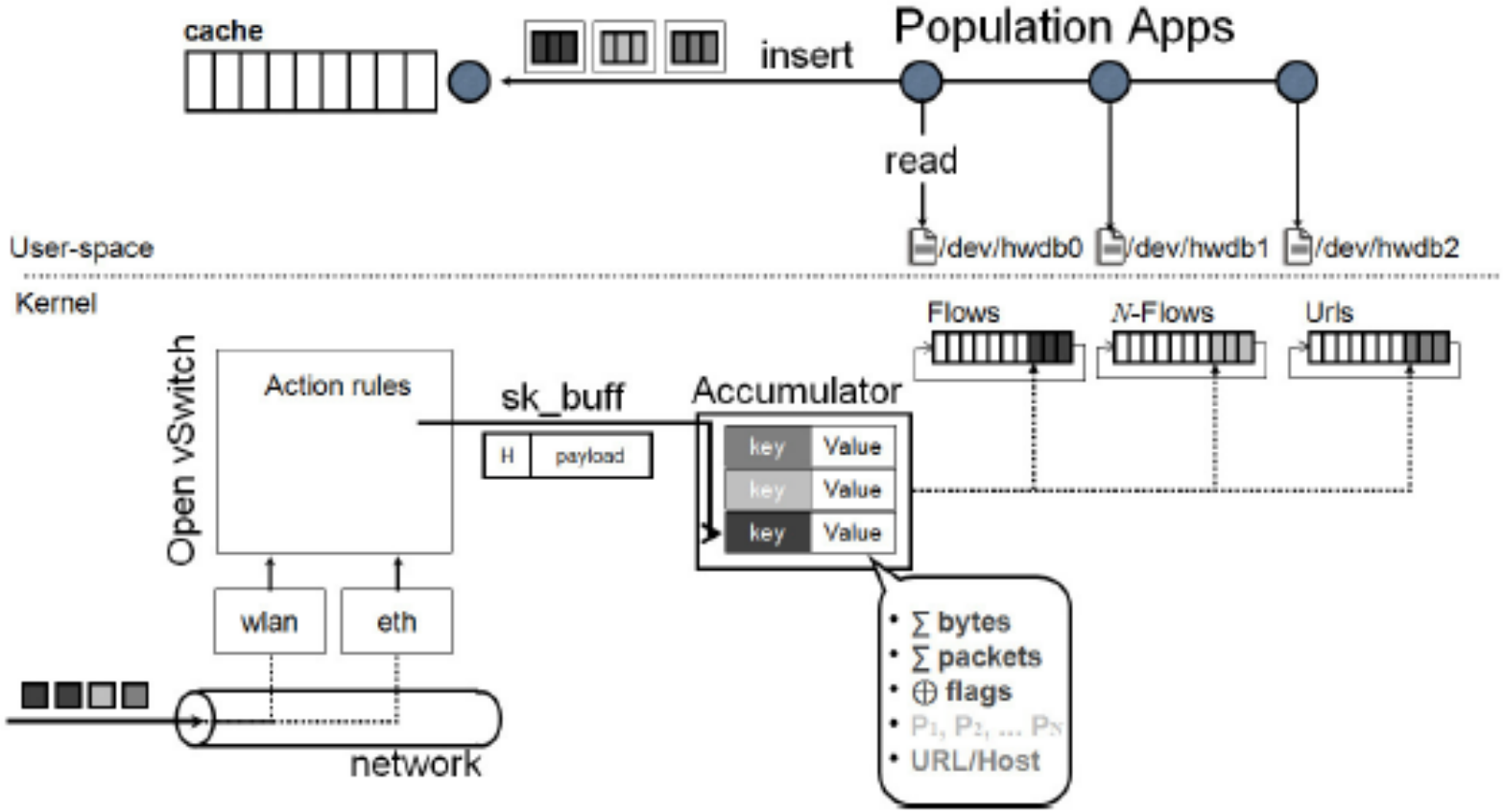
Monitoring Applications

- Population – measure some aspect of the system and insert the resulting measurement data into the Cache.
- Persistence – extract data from the Cache in order to store that data, or information derived from that data, into the persistent time-series database and/or the persistent relational database.
- Reaction – register interest in particular behaviour patterns; when such a pattern is detected by the Cache, it notifies the application of the occurrence to enable it to react to the event.
- Display – extract real-time data from the Cache for display to a user of the system. These applications may also extract data from the persistent databases, if necessary.

Tables in the Homework cache

- **Ephemeral**
  - **Flows** (proto, saddr, sport, daddr, dport, npkts, nbytes)
  - **Links** (macaddr, rssi, nretries, npkts, nbytes)
  - **UserEvents** (application, logtype, logdata)
  - **Sys** (message)
  - **Urls** (proto, saddr, sport, daddr, dport, host, URI, cnt)
- **Persistent**
  - **Leases** (macaddr, ipaddr, hostname, action)
  - **Devices** (macaddr, status)
  - **Allowances** (ipaddr, bytes)
  - **BWUsage** (ipaddr, bytes)

Logging architecture



Raw event generation

- **Link** information obtained using libpcap (RadioTap header)
- An additional action in openvswitch passes each packet to a kernel accumulator, which accumulates the following data:
  - Flow records
  - Data about the first N packets in each flow
  - For HTTP packets, the HTTP request header
- A once per second timer interrupt causes the kernel accumulator to write accumulated records to three different devices:
  - /dev/hwdb0 returns flow accumulations (to insert into table **Flows**)
  - /dev/hwdb1 has statistical information about the first N packets (currently, N = 10) of each flow
  - /dev/hwdb2 has http request headers to insert into table **Urls**
- Population applications simply have reads outstanding on these devices; when their reads are satisfied, they format insert commands into relevant tables and then call the Cache
- **Lease** information is inserted into the Cache by the DHCP module

University of Glasgow

Detecting Patterns

HOMEWORK

- Events of interest may involve complex combinations of the raw measurement data that is moving through the pub/sub topics and persistent relations
- We have defined the Glasgow Automaton Programming Language (GAPL) for specifying automata; these automata provide an imperative mechanism for detecting complex patterns.
- A reaction application registers an automaton against the database; the automaton sends events to the application when such events are detected

Form of an automaton

HOMEWORK

automaton::= subscriptions behavior
| subscriptions declarations behavior
| subscriptions declarations initialization behavior
| subscriptions associations behavior
| subscriptions associations declarations behavior
| subscriptions associations declarations initialization behavior

subscription::= “subscribe” <localvar> “to” <TopicName> “;”

association::= “associate” <localvar> “with” <PersistentTableName> “;”

declaration::= variabletype variablelist “;”

Simple examples

HOMEWORK

Process-based pub/sub broker

```
subscribe e to <topic>;
behavior {
        send(e);
}
```

Stream merge

```
subscribe s to S;
subscribe t to T;
behavior {
        if (currentTopic() == 'T')
                publish('U', t);
        else
                publish('U', s);
}
```

Simple Automaton

```
subscribe u to Urls;

map url;

initialization {
        url = Map(int);
        insert(url, Identifier('www.google.com'), 0);
        # insert others here
}

behavior {
        if (hasEntry(url, Identifier(u.hst))) {
                send(u.saddr, u.hst);
        }
}
```

University of Glasgow

Implementation

HOMEWORK

- Compiler generates instructions for stack machine
- Each compiled automaton is bound to a separate thread
- When a tuple is inserted into a Table, each automaton thread that has subscribed to that topic is given access to that tuple and awakened
- Upon being awakened, the automaton executes its behavior clause
- If the automaton executes a “send” procedure call, this will result in the arguments being sent as an RPC to the registered reaction application
- All heap storage associated with local variables is reference-counted.

University of Glasgow

Language features

HOMEWORK

- Support for maps, sequences, windows, iterators, identifiers
- Ability to associate relational table with a map
- Ability to inject additional events into pub/sub channels
- Ability to send events to registered processes
- Access to timer pub/sub channel

University of Glasgow

Basic Data Types

HOMEWORK

Type	Description
int	64-bit signed integer
real	double-precision floating point
tstamp	64-bit unsigned integer (ns since the epoch)
bool	true or false
string	variable-length UTF8 array

Aggregate/Supporting Data Types

HOMEWORK

Type	Description
sequence	ordered set of heterogeneous data type instances
map	map from an identifier to an instance of the bound type
window	collection of bound type instances that is constrained either to a fixed number of items or a fixed time interval
identifier	key used in maps
iterator	used to iterate over all instances in a map (keys) or window (data values)

University of Glasgow

Interpreter functions

HOMEWORK

Map functions
map Map(map.type)
void insert(map, identifier, instance)
void remove(map, identifier)
map.type lookup(map identifier)
bool hasEntry(map identifier)

Window functions
window Window(win.type, winconstr, constrval)
void append(window, instance[, tstamp]
real average(window)
real stdDev(window)

University of Glasgow

Interpreter functions (cont)

HOMEWORK

Iterator functions
iterator Iterator(map | window)
bool hasNext(iterator)
identifier next(mapIterator)
win.type next(winIterator)

Miscellaneous functions
void send(basictype | sequence | window[, ...])
void publish(topic basictype | sequence[, ...]
string currentTopic()
identifier Identifier(basictype[, ...]
string String(basictype[, ...])
void destroy(aggregate.type)

More Sophisticated Example

```
subscribe f to Flows;
associate a with Allowances;
associate b with BWUsage;
int n, limit;
identifier ip;
iterator it;
sequence s;
string st;
behavior {
        ip = Identifier(f.daddr);
        if (hasEntry(a, ip)) {
                limit = seqElement(lookup(a, ip), 1);
                if (hasEntry(b, ip))
                        n = seqElement(lookup(b, ip), 1);
                else
                        n = 0;
                n += f.nbytes;
                s = Sequence(f.daddr, n);
                if (n > limit)
                        send(s, limit, 'limit exceeded');
                insert(b, ip, s);
        }
}
```

Implementation details

- Cache runs as multi-threaded server on Linux, Cygwin, OSX
- Circular buffer for ephemeral tables configured to retain ~2 hours worth of raw data
- Architecture was particularly focused on minimizing critical sections between automata threads
- Significant effort expended in performance engineering
- The following slides show:
  - execution costs for interpreter operations
  - scheduling delays as a function of number of subscribed automata and as a function of insertion rate
  - maximal insertion rate as a function of size of table tuples

Performance

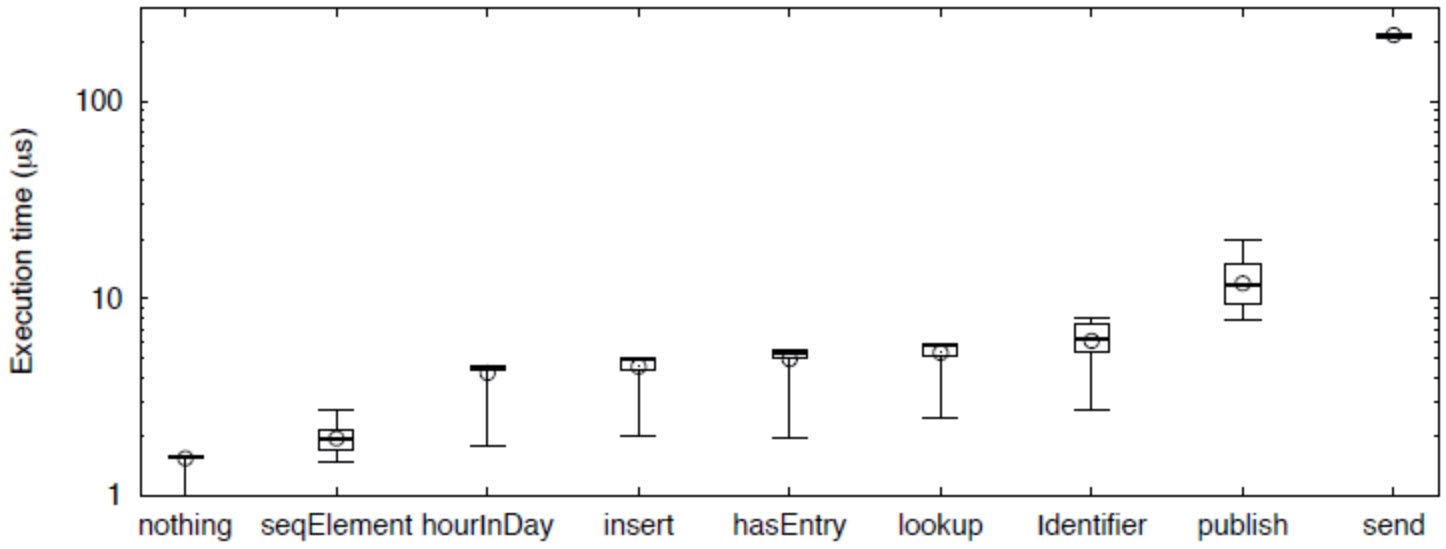
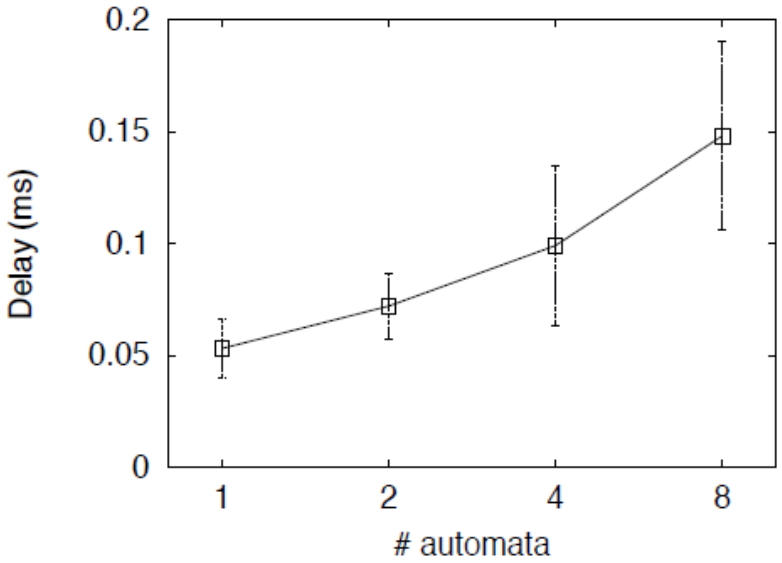


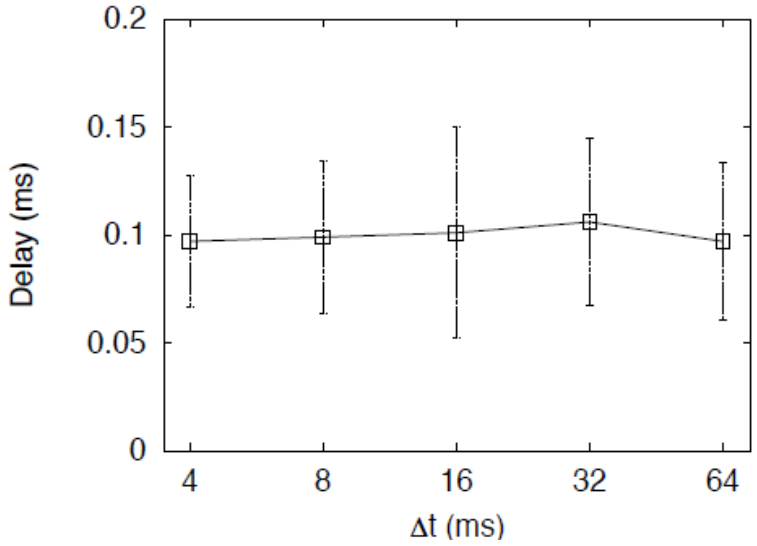
Fig. 7. The execution cost of built-in functions

Performance (cont)

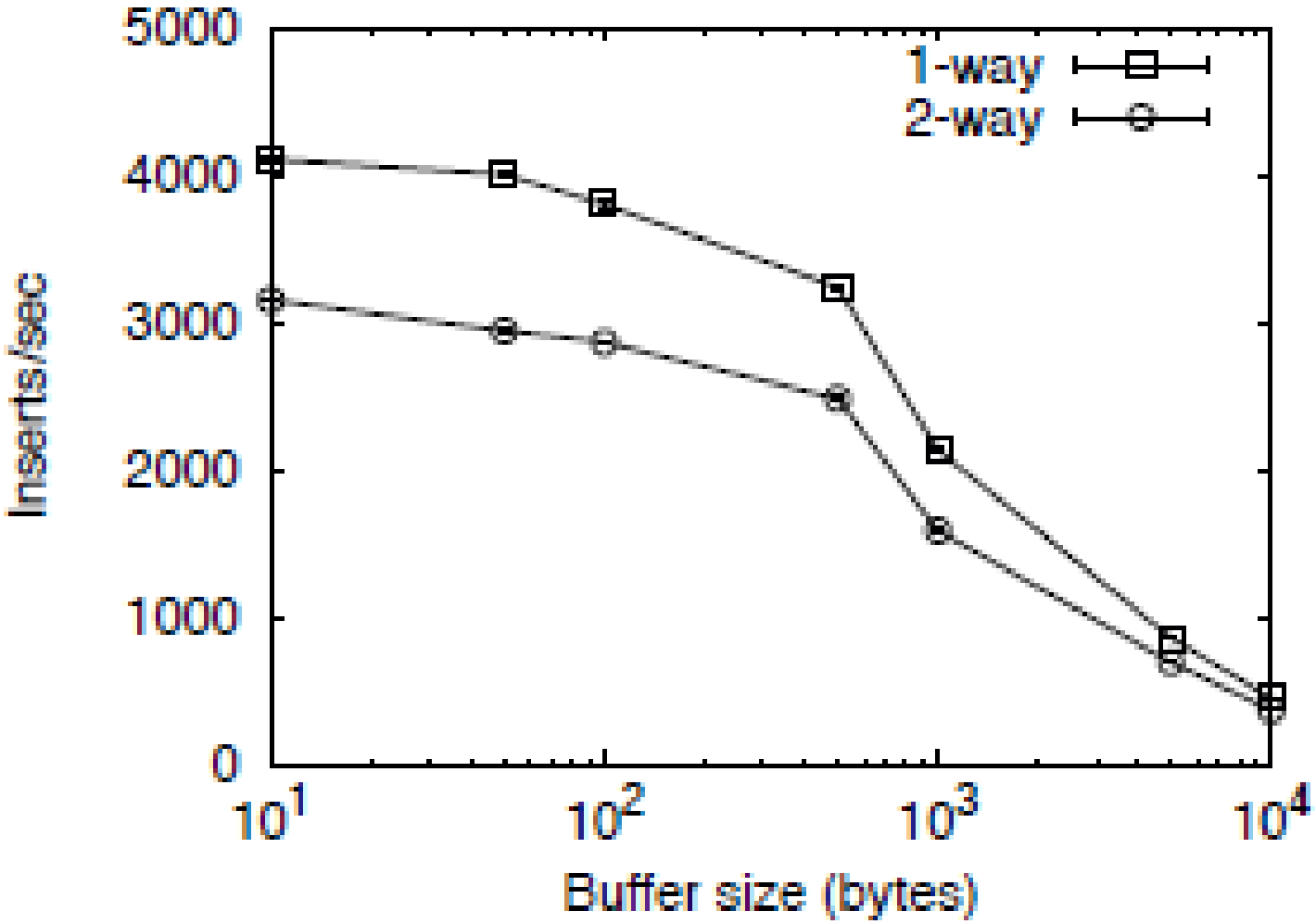
Delay vs # automata, Δt = 8ms



Delay vs arrival rate, 4 automata

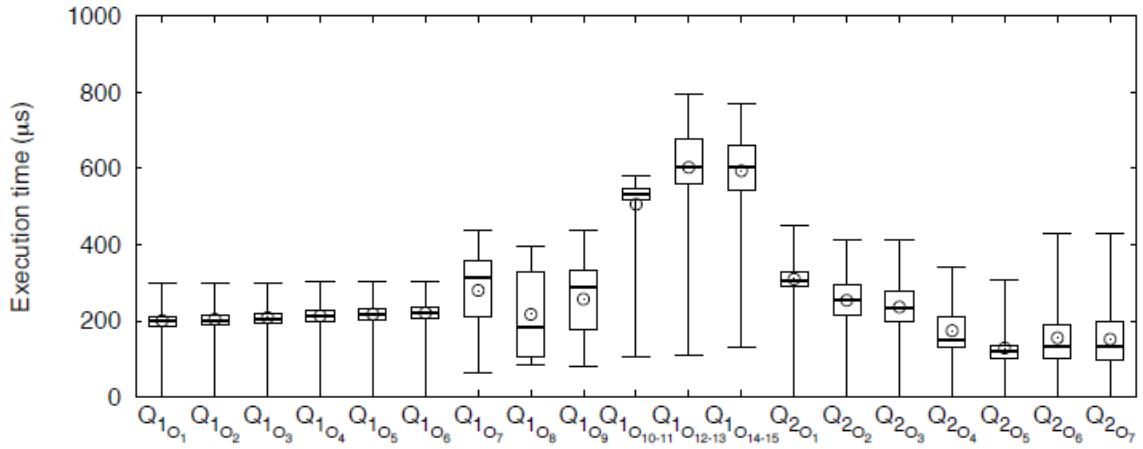


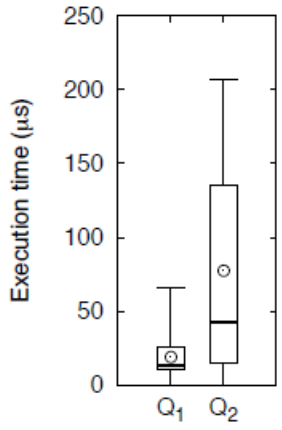
Performance (cont)



Why should you care?

1. The similarity of windows to events over streams enables the implementation of several different solutions using differing numbers of automata and topics.



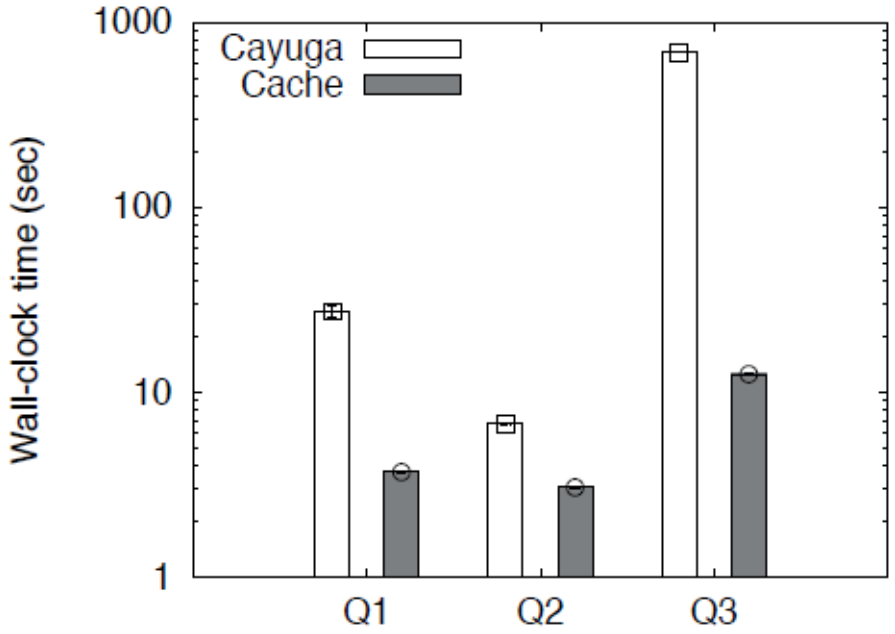


Why should you care (2)?

2. The architecture of the information plane, coupled with the pub/sub architecture within the Cache, provides significant flexibility in how one designs the monitoring system.
3. The system has been shown to be immediately applicable to a number of application domains (network monitoring, factory control, stock analysis)

Why should you care (3)?

- Comparison against Cayuga
  1. SELECT * from Stocks PUBLISH T
  2. Look for double-top formation in the price chart
  3. Detect continuous runs of increasing prices for each stock



University of Glasgow

Conclusions

HOMEWORK

- The unification of publish/subscribe and stream database concepts has enabled us to address a number of complex event processing scenarios in several application domains
- Glasgow automata permit one to specify custom operators dynamically
- Automata are at an intermediate level of abstraction between “one-liners” and low-level implementations
- Thus, GAPL permits customization of queries, consistent with streams and relations, to achieve different performance requirements.

University of Glasgow

Future Work

HOMEWORK

- Mapping directly from “one-liners” (e.g. Cayuga) to GAPL source or instructions for its stack machine
- JIT code generation from stack machine byte codes to hardware instructions
- Establish the scalability of independent automata with increasing number of cores
- Apply to other domains
- Network of Cache instances
- Release of Cache through open source

Questions?