Complex Event Processing: DSL for High Frequency Trading

Richard Tibbetts
StreamBase Systems
QCon London 2011
Myth: High level domain specific languages are too slow for HFT.

Reality: High level domain specific languages can deliver better performance than system programming languages when tailored to a specific task.
High Frequency Trading

- Financial trading where latency is critical to profitability

- Four Main Scenarios
  - Alpha seeking – arbitrage
  - Rebate seeking – market making
  - Transaction cost minimization – execution management
  - Service providers – risk management, exchanges

- Different tolerances for latency across asset class, use cases
  - Speed to catch opportunity, speed to not get run over, speed to keep customers

- Most often from scratch in systems programming languages
  - C++, maybe Java

- Lots of talk and some use of hardware acceleration, FPGA and GPUs
Complex Event Processing aka Event Processing

- **Software organized by events (compare object oriented)**
  - What’s an event? What’s an object?
  - And event is something can trigger processing, can include data.
  - Naturally but not usually represents a “real world” event or observation.

- **Complex Event Processing Platforms**
  - Software stack for event based systems, event driven architectures
  - Event Programming Language – SQL-based, Rules-based, or State-based
  - Commercial and open source: StreamBase, Progress, Microsoft, IBM, Oracle, SAP, Esper, Drools and many more

- **Adopted in financial services and other markets**
  - System monitoring, industrial process, logistics, defense/intelligence

- **Other Event Processing Approaches:**
  - Erlang, Actors, node.js, .NET Rx
Why a DSL?

- High level
- Graphical
- Appropriate for purpose
- Understandable
- Flexible

“Simplicity is always disruptive”
- Clayton Christensen

“We developed in 4 months what would have taken 4 years.”
- StreamBase customer Kaeos

“We modify the behavior of our trading system every day.”
- StreamBase customer PhaseCapital
Challenges for DSL in HFT

- **Ultra Low Latency**
  - Sub-millisecond is standard, sub-100-micro is desired.

- **Large Data Volumes**
  - Hundreds of thousands of quotes per second, thousands of orders

- **Demanding Operational Environment (in some ways)**
  - Not 24x7, not low touch, but availability during market hours is key

- **Sophisticated Data Processing (sometimes)**
  - Options pricing, yield curves, risk metrics and more

Agenda

- Intro: Myth, Reality, HFT, CEP
- Benefits of a DSL, Challenges of HFT
- StreamBase Accomplishments – Performance and Productivity
- Designing a Language for HFT: Performance and Extensibility
  - Static Analysis
  - Code generation and the Janino compiler
  - Garbage optimization
  - Adapter API, FIX Messaging
  - Parallelism, lanes and tiers
  - Integrations, C++ and Java plugins
- Lessons Learned
- Shameless Plug
- Acknowledgements, Questions and Answers
StreamBase Event Processing Platform

Developer Studio
Graphical StreamSQL for developing, back testing and deploying applications.

Input Adapter(s)
Inject streaming (market data) and static (reference data) sources.

StreamBase Server
High performance optimized engine can process events at market data speeds.

Output Adapter(s)
Send results to systems, users, user screens and databases.

Studio Integrated Development Environment

Visualization
StreamBase StreamSQL EventFlow

- Rapid Deployment & Unit Testing
- Modularity & Polymorphism
- Off The Shelf Connectivity
- Interfaces & Extension Points
- Off The Shelf Business Logic
Kinds of Applications

Data Management
- Aggregation
- Cleansing
- Normalization
- Symbology matching
- Latency monitoring, alerting

Back testing
- Market replay
- Chaining multiple sources

Data Enrichment
- Analytics
- Derived calculations
- Index computation

Risk
- Pre & post trade
- Position keeping
- Auto hedging

System Monitoring
- Feed monitoring
- Orders

Alpha Seeking
- Trading strategies
- Buy/sell signals

Dealing
- Market Making
- Pricing
- Auto quoting

Trade Execution
- Execution algorithms
- Smart order routing
- TCA
StreamBase Accomplishments Performance & Productivity

- **Productivity**
  - Don’t reinvent the wheel… or the gaskets, fuel tank, seats, air bag
  - Connectivity (100+ adapters), plumbing, scalability built in
  - Support for agile development process
    - Quants and developers working together
  - Decrease time-to-market and time-to-change 40-90%
    - 10 weeks to 2 days
    - Improved communication, iterative development, business alignment

- **Performance**
  - Ultra low latency – As low as 80 microsecond end to end latency
  - Predictable latency – 99th percentile, minimize outliers
  - High throughput – 100s of thousands of messages per second per core
  - Scale – Horizontally and Vertical, Multi-core and Cluster
How did we do it?

- **Compilation and Static Analysis**
  - Design the language for it

- **Modular abstraction, interfaces**
  - Quants and Developers Collaborate

- **Bytecode generation and the Janino compiler**
  - Optimized bytecodes, in-memory generation

- **Garbage optimization**
  - Pooling, data class, invasive collections

- **Integrations, C++ and Java plugins**
  - Efficient native interfaces

- **Adapter API, FIX Messaging**
  - Threading and API structure for ultra low latency

- **Parallelism, Clustering, Lanes and Tiers**
  - Scalability with latency in mind

- **Named Data Formats, Schemas**
  - Sharing data and semantics between apps
Compilation Static Analysis

- **Design the language for compilation and performance**
  - Static typing, controlled mutation

- **Graphical structure is natural for domain**
  - Statically defined application traversal pattern

- **Avoid listeners, virtual/dynamic dispatch, registration**
  - Except where necessary for extensibility, parallelism

- **Graph defines data sharing**
  - Pure functional expression language
  - Immutable messages, shared mutable data in tables
Modular abstraction, interfaces

- Heterogeneous Teams: Quants and Developers Collaborate
- Graphical Language still requires sophisticated abstraction
  - Modules, parameterization, polymorphism, hygienic macros
- Interfaces support dependency injection
- Back testing and production deployment of same code
  - Back testing harness uses same interfaces, historical data
- Allow reuse of infrastructure components across asset classes
  - Order state management, book building, etc
Bytecode generation and the Janino compiler

- **Composite data types**
  - Composed of primitive Java data types, arrays

- **Explicit inlining**

- **Monomorphic call sites**

- **Work with the JIT**

- **Calling convention**
  - Introduction of dataclass
  - Queue structures

```java
class Module_fooBar extends MainModule {
    public void enqueueTuples(StreamProperties stream, byte[] buffer) {
        if (stream.matchesIn()) {
            // demarshall tuples from the wire, and call s__ln(...).
        }
    }
    void s__ln(int f1_value, boolean f1_null, byte[] f2_data, long f2_offlen) {
        op__Where(f1_value, f1_null, f2_data, f2_offlen);
    }
    void op__Where(int f1_value, boolean f1_null, byte[] f2_data, long f2_offlen) {
        if (f1_null && f1_value > 5) {
            op__Select(f1_value, f1_null, f2_data, f2_offlen);
        }
    }
    void op__Select(int f1_value, boolean f1_null, byte[] f2_data, long f2_offlen) {
        int x_value; boolean x_null; byte[] y_data; long y_offlen;
        if (f1_null) {
            x_value = 0; x_null = true;
        } else {
            x_value = f1_value * 2; x_null = false;
        }
        if (f2_offlen == OFFLEN_NULL) {
            y_data = null; y_offlen = OFFLEN_NULL;
        } else {
            y_data = EvalUtil.concat(f2_data, offlen, EvalUtil.stringToBytes(\"m + f1_value\"));
        }
        s__Out(x_value, x_null, y_data, y_offlen);
    }
    void s__Out(x_value, x_null, y_data, y_offlen) {
        // send output to any subscribers
    }
    ...
```
Garbage optimization

- All objects live forever or highly transient
- Minimize per-event transient objects (to zero)
  - Test harness to measure per-event garbage
- Collector tuning
  - Smaller heaps, smaller young gen, faster promotion for low latency
  - Clustering for large apps in small heaps
- Primitive data types, infrequently allocated arrays
- Test harness for identifying garbage sources

[Image of recycling symbol]
DataClass

- A shared struct, all users of the data object inject members

Compiled Tuple Implementation

- Efficient access
- Minimize copying, mutation

Invasive Collections

- Invasive collections add their own members to DataClass
- No header objects

```java
/**
 * Add a field that will be managed by DataClass.
 */
public int addManagedField(final CFieldDecl field) {
    if (state != STATE_CONSTRUCTING) {
        throw new IllegalStateException("Can only add managed field");
    }
    managedFields.add(field);
    return managedFields.size() - 1;
}

/**
 * Add a field that will not be managed by DataClass.
 */
public void addField(final CFieldDecl field) {
    clazz.add(field);
}

public void addFields(final CFieldDecl[] decls) {
```
Integrations, C++ and Java plugins

- **Efficient native interfaces**
- **JNI does integers and byte buffers. No objects**
  - Tailor C++ APIs to this reality
  - Infrequent resizing
- **Java APIs designed for garbage efficiency**
  - Primitive types
  - Object reuse
Adapter API, Third Party Integrations

- Threading and API structure for ultra low latency
- Adapter threads carry the message through application processing
- Single thread from ingest to output
  - Requires care to avoid deadlocks in third party libraries
- Memory management hints in API: reuseTuple, factory methods
- Compiled tuple implementation – backed by dataclass
- Tightly integrate key messaging technologies
  - FIX: QuickFIX, Cameron, etc
  - Venues, Hardware acceleration
  - Cluster Messaging: P2P, Solace
Parallelism, Clustering, Lanes and Tiers

- Scalability with latency in mind
- For low latency, single machine per message, minimize queues
- Parallelize in the middleware, e.g. Solace
- Lanes offer stable latency when scaling, less efficient hardware utilization
- Tiers for efficiency of node-role
Named Data Formats, Schemas

- Data formats are key driver of event driven app design
- Named schemas for sharing data types, fields, definitions
- Basis for Interfaces and Extension Points
  - Teams combining developers, quants, analysts
- Non-Flat message model (despite SQL heritage)
  - Sub-tuples, Lists
Lessons Learned, What Not To Do

- **Messages are fatter than you would think**
  - Particularly internal messages; often have 100-200 fields

- **Overuse of code generation (passive voice)**
  - Not everything needs to be hyper-optimized
  - Favor active voice code, with active voice tests, and passive voice subclasses. Trust in monomorphic call sites and the JIT. But verify.

- **Delayed emphasis on separate compilation**
  - Formalize and test calling convention early

- **Invest in performance measurement**
  - Don’t be afraid to have your core engineers writing performance measurement and analysis harnesses
Shameless Plugs

- **StreamBase**
  - You could build one of these yourself, or use ours...
  - Download and test out the full product [http://www.streambase.com](http://www.streambase.com)
  - Build something and submit to the StreamBase Component Exchange
    - [http://sbx.streambase.com](http://sbx.streambase.com)
  - Contact us to buy or to an OEM partner, offices London, Boston, New York
  - We’re hiring
  - We’re training
    - [http://www.streambase.com/developers-training-events.htm](http://www.streambase.com/developers-training-events.htm)

- **DEBS – Distributed Event Based Systems**

- **EPTS – Event Processing Technology Society**
  - [http://ep-ts.org](http://ep-ts.org) industry consortium
Questions?

Download StreamBase and More Information
http://www.streambase.com