Noname manuscript No.
(will be inserted by the editor)

Faster FAST: Multicore Acceleration of Streaming Financial Data

Virat Agarwal · David A. Bader · Lin Dan · Lurng-Kuo Liu · Davide Pasetto · Michael Perrone · Fabrizio Petrini

Received: date / Accepted: date

Abstract By 2010, the global options and equity markets will average over 128 billion messages per day, amounting to trillions of dollars in trades. Trading systems, the backbone of the low-latency high-frequency business, need fundamental research and innovation to overcome their current processing bottlenecks. With market data rates rapidly growing, the financial community is demanding solutions that are extremely fast, flexible, adaptive, and easy to manage. This paper explores multiple avenues to deal with the decoding and normalization of Option Price Reporting Authority (OPRA) stock market data feeds encoded with FIX Adapted for Streaming (FAST) representation, on commodity multicore platforms, and describes a novel solution that encodes the OPRA protocol with a high-level description language. Our algorithm achieves a processing rate of 15 million messages per second in the fastest single socket configuration on an Intel Xeon E5472, which is an order of magnitude higher than the current needs of the financial systems. We also present an in-depth performance evaluation that exposes important properties of our OPRA parsing algorithm on a collection of multicore processors.

1 Introduction

A typical market data processing system consists of several functional units that receive data from external sources (such as exchanges), publish financial data of interest to their subscribers (such as traders at workstations), and route trade data to various exchanges or other venues. This system, known as a *ticker plant*, is shown in Figure 1a. Examples of functional units include feed handlers, services management (such as permission, resolution, arbitration, etc.), value-added, trading system, data access, and client distribution. The feed handler is the component that directly interacts with the feed sources for handling real time data streams, in either compressed or uncompressed format, and decodes them converting the data streams from source-specific format into an internal format, a process called *data normalization*. According to the message structure in each data feed, the handler processes

Virat Agarwal, Lin Dan, Lurng-Kuo Liu, Michael Perrone, Fabrizio Petrini, IBM TJ Watson Center, Yorktown Heights, NY 10598 USA, E-mail: {viratagarwal,ldanin,lkliu,mpp,fpetrin}@us.ibm.com

Davide Pasetto, E-mail: pasetto_davide@ie.ibm.com
IBM Computational Science Center, Damastown Industrial Estate, Mulhuddart, Dublin (Ireland).

David A. Bader, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA.

each field value with a specified operation, fills in the missing data with value and state of its cached records, and maps it to the format used by the system. The ability of a feed handler to process high volume market data stream with low latency is critical to the success of the market data processing system, and is the main focus of this paper.

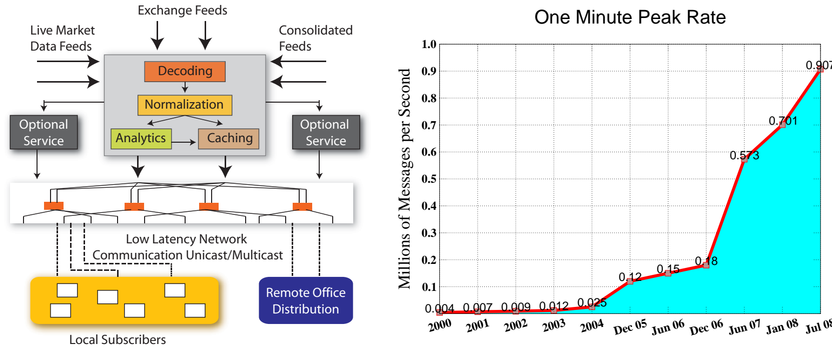

Fig. 1: (a) High-Level Overview of a Ticker Plant, (b) OPRA market peak data rates

1.1 Skyrocketing Data Rates

The Options Price Reporting Authority (OPRA) is the securities information processor that disseminates, in real-time on a current and continuous basis, information about transactions that occurred on the options markets. It receives options transactions generated by participating U.S. exchanges, calculates and identifies the "Best Bid and Best Offer", consolidates this information and disseminates it electronically to the financial community. Fueled by the growth of algorithmic and electronic trading, the global options and equities markets are expected to produce an average of more than 128 billion messages/day by 2010, rising from an average of more than 7 billion messages a day in 2007 [12]. In the options market, the OPRA consolidated feed for all US derivatives business represents a very significant portion of market data in the national market system. Figure 1b shows that the OPRA market data rates have dramatically increased over the course of the past 4 years, approaching a peak of 1 million messages per second. The traffic projection for OPRA alone is expected to reach an average of more than 14 billion messages/day in 2010 [7]. As market data rates continue to skyrocket, high speed, low latency, and reliable market data processing systems are becoming increasingly critical to the success of the financial institutions both in the U.S. and abroad.

1.2 High Performance Computing in the Data Center

It is important to note that low latency is critical, especially with the increasing competition as well as diminishing profit margin. How fast a trading system can respond to the market will determine who wins and who loses, even a few milliseconds gap in latency is enough to make the difference. Not surprisingly, many of the technologies that are commonly used in high performance computing are rapidly appearing in the data centers of many financial institutions [4, 3, 1]. Together with the increased performance, data centers are also encountering typical problems in high performance computing: the complexity of developing, testing and validating parallel software. The need is to reduce power consumption in the processing units that are often "co-located" near the data feeds to minimize the communication latency and reduce floor space requirement, which typically comes at a high premium. For these

reasons, the financial community is demanding solutions that are extremely fast, easy to program, and adaptable to dynamically changing requirements.

1.3 Contribution

In this paper we explore several avenues to implement an OPRA FAST decoder and data normalization on commodity multicore processors. We have first attacked the problem by optimizing a publicly available version of an OPRA decoder, and based on this preliminary implementation we have developed two more versions of the decoder, one that has been written from scratch and hand-optimized for multicore processors and one that uses DotStar [10], a high-level protocol parsing tool that we have recently designed and implemented. The paper provides four main contributions. (1) The implementation of a high-speed hand-optimized OPRA decoder for multicore processors. In the fastest configuration, this decoder achieves an impressive processing rate of 14.6 (3.4 per thread) millions messages per second with a single Intel Xeon E5472 quad-core socket. (2) A second implementation, based on the DotStar protocol processing tool is described in Section 4, that is able to capture the essence of the OPRA structure in a handful of lines of a high-level description language. This is combined with the same set of building blocks (actions) described above, triggered by the protocol scanner. The DotStar protocol parser is comparable with the hand-optimized parser, on average only 7% slower on 5 distinct processor architectures. In the fastest single-socket configuration, a single Intel Xeon E5472 quad-core is able to achieve a rate of 15 million messages per second. (3) An extensive performance evaluation that exposes important properties of the DotStar OPRA protocol and parser, and analyzes the scalability of five reference systems, two variants of Intel Xeon, AMD Opteron, the IBM Power6, and the SUN UltraSPARC T2. (4) Finally, we provide insight onto the behavior of each architecture trying to explain *where* the time is spent by analyzing, for all the processor architectures under evaluation, each action associated to DotStar events. All the action profiles are combined in a cycle-accurate performance model, presented in Section 6, that helps determine the optimality of the approach, and to evaluate the impact of architectural or algorithmic changes for this type of workload. We believe that this level of accuracy can be useful to both application developers and processor designers to clearly understand how these algorithms map to specific processor architectures, allowing interesting what-if scenarios.

One evident limitation of this work is that it addresses only a part of the feed handler, leaving many important questions unanswered. For example, the network and the network stack are still areas of primary concern. Nevertheless, we believe that the initial results presented in this paper are ground-breaking because they show that it is possible to match or exceed speeds that are typical of specialized devices, such as FPGAs, using commodity components and a high-level formalism that allows quick configurability.

2 OPRA feed decoding

OPRA messages are delivered through the national market system with a UDP-based IP multicast. These messages are divided into 24 data lines or channels (48 when counting redundant delivery) based on their underlying symbol. Multiple OPRA messages are encapsulated in a block and then inserted in an Ethernet frame. The original definition of an OPRA message is based on an ASCII format, which uses string based encoding and contains redundant information. With the growth of volume, a more compact representation for messages was introduced: OPRA FAST (FIX Adapted for STreaming).

The techniques used in the FAST protocol include implicit tagging, field encoding, stop bits, and binary encoding. Implicit tagging eliminates the overhead of field tags transmission. The order of fields in the FAST message is fixed and the meaning of each field can be determined by its position in the message. The implicit tagging is usually done through the XML-based FAST template. The presence map (PMAP) is a bit array of variable length at the beginning of each message where each bit is used to indicate whether its corresponding field is present. Field encoding defines each field with a specific action, which is specified in a template file. The final value for a field is the outcome of the action taken for the field. Actions such as “copy code”, “increment”, and “delta” allow FAST to remove redundancy from the messages. A stop bit is used for variable-length coding, by using the most significant bit in each byte as a delimiter. This limits the amount of useful information to 7 bits for every byte of data. FAST uses binary format to represent field values.

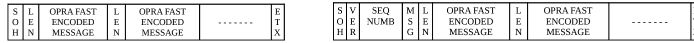

Fig. 2: OPRA FAST encoded packet format. (a) Version 1.03 (b) Version 2.0

Figure 2a shows the format of an encoded OPRA version 1.03 packet. Start of Header (SOH) and End of Text (ETX) are two control characters that mark the start and end of the packet. In OPRA FAST version 2.0 (Fig. 2b) there is a header after SOH and before the first message. The first byte of an encoded message contains the length in bytes and is followed by the presence map, e.g., presence map 01001010 means field 1, field 4 and field 6 are present, counting from left. The presence map field is followed by several variable length integer and string fields that contain the data encapsulated by the message. Data fields that are not present in an encoded message but that are required by the category have their value copied from the same field of previous messages and optionally incremented. OPRA data fields can be unsigned integer or string.

We use the reference OPRA FAST decoder provided along with the standard. This implementation starts by creating a new message, parses the presence map, computes its length by checking the stop bit of every byte until one is found set, masks the stop bit and copies all the data into temporary storage. The presence map bits are then examined to determine which fields are present and which require dictionary values. The implementation proceeds by checking the category of the message and calls a specific decoder function, which implements the actual decoder for each field in the message. We initially tried to optimize the reference decoder using a *top-down* approach, by speeding up the functions that are more time-consuming. Unfortunately the computational load is distributed across a large number of functions, and our effort resulted in a very limited performance improvement.

3 A Streamlined Bottom-Up Implementation

We identified primary compute intensive kernels, such as processing variable length encoded presence map (PMAP), integer and string fields, identifying field index, and category dependent field copy, from the decoding algorithm. We optimize these kernels using various techniques: unroll to efficiently utilize instruction pipelines, analyze the instructions from assembly level and minimize average processor cycles consumed per byte of input data.

The reference implementation decodes the input stream by first identifying the category of the input message and then calls a category specific routine to process subsequent data fields. This leads to a very branchy code. Our implementation replaces the category-specific routines by category-specific bitmaps. The number of bits in this bitmap is equal to the total number of fields contained in the OPRA protocol specification. In the current version of this

specification the bitmap can be represented by a 64 bit variable. The index of the bit in the bitmap specifies the field it represents, and only bits corresponding to fields required by the category are set. Once the category of the message is determined, this bitmap is passed to the next building block for specifying the fields that are relevant to a message of this category.

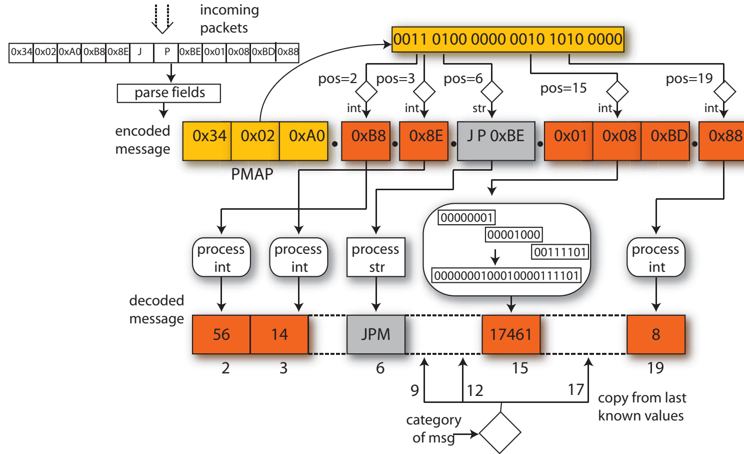

Fig. 3: Presence and field map bit manipulation.

Another optimization is related to processing the PMAP field. The reference code processes this field bit by bit, testing every bit to check if it set or not. However, since a subset of fields are required for each message category, and given that only a fraction of bits are typically set in an incoming message, it becomes very inefficient to process each PMAP bit. We use PMAP to specify the data fields that are contained in the incoming message, that need to be processed and updated in the decoded format. An *xor* operation between this PMAP and the category-specific bitmap gives the fields that are relevant to the incoming message but are not present. We call this the *copy* PMAP, that specifies the fields that need to be copied from last received values.

To determine the next field present in the incoming message we count leading zeroes (*clz*) from the PMAP field. This bit (in PMAP) is set to zero after the data field is processed to enable repetition of the *clz* operation for subsequent data fields. The *clz* operation is also used for the *copy* PMAP to determine the indexes of the fields that are required to be copied from previous values. Figure 3 illustrates our optimized decoding algorithm.

We employ another optimization for parsing string fields. To maintain an efficient data structure we maintain a separate buffer for string values. The strings from the incoming message are copied into the buffer and the decoded message only contains the offset of the string from this buffer. This increases the efficiency of the decoding algorithm because, rather than replicating the entire string, we only store the string offset.

These optimizations create efficient high performance kernels for basic decoding tasks and increase the elegance of the design. This *bottom-up* approach eliminates the complex control flow structure that exists in the reference implementation. In the next section we will discuss how these kernels can also be integrated into a high level protocol parser.

4 High-Level Protocol Processing with DotStar

A fundamental step in the semantic analysis of network data is to parse the input stream using a high-level formalism. A large amount of work approached these issues by using

declarative languages to describe data layout and transfer protocols. Examples are Interface Definition Languages (IDL), such as XDR [11], RPC [11], and regular languages and grammars, like BNF, ABNF [8], lex and yacc; or declarative languages like binpac [9]. These solutions are often tailored for a specific class of protocols, like binary data or text oriented communication. A common characteristic of all these solutions is that they are very high level and "elegant", and provide the user with great expressiveness, but the generated parser performance is often one order of magnitude slower than an equivalent handcrafted one, and may be unable to parse real-time heavy-volume applications.

While exploring how to extend our fast keyword scanner automaton to handle regular expression sets, we developed DotStar [10], a complete tool-chain that builds a Deterministic Finite Automaton (DFA) for recognizing the language of a regular expression set. The automaton is an extension of the Aho-Corasick [6], the *de facto* standard for fast keyword scanning.

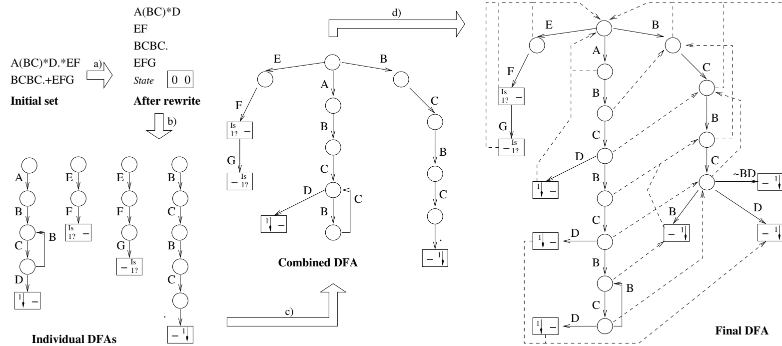

Fig. 4: A graphical representation of DotStar compiler steps: a) is the pre-processor, which reduces the number of states, step b) builds individual automata for each regular expression, c) combines them in a unique automaton and d) computes the failure function.

Finding every instance of a regular expression pattern, including overlaps, is a "complex" problem both in space and time [5]. Matching a complete set of regular expressions adds another level of complexity: DotStar employs a novel mechanism for combining several regular expressions into a single engine while keeping the complexity of the problem under control. DotStar is based on a sophisticated compile time analysis of the input set of regular expression and on a large number of automatic transformations and optimizing algorithms. The compilation process proceeds through several stages (see Figure 4): at first simplifies each regular expression in a normal form, rewrites it and splits it into sub-expressions and transforms into a Glushkov [2] Non-deterministic Finite Automaton (NFA) and then turns it into a DFA, These automata are combined together by an algorithm that operates on their topology. The resulting graph is extended, as in the Aho-Corasick algorithm, by a "failure" function, that can further modify the graph structure. The result is a single pass DFA that: (1) groups all regular expression in a single automaton, (2) reports all matches, including every overlapping pattern, and (3) it is as memory efficient as an NFA.

We defined a simple declarative language, called DSParser, that describes a data layout as a sequence of regular expression fragments "connected" using standard imperative constructs, like if/then/else/while. Actions can be inserted after a (partial) expression is recognized; these actions are either system actions, that perform common operations or user defined functions on blocks of input data. A precompiler tool parses the declarative language and builds a suitable regular expression set that is compiled using the DotStar toolchain.

A small number of states of the resulting automaton is then annotated with the system and user defined actions specified in the initial protocol definition. The DSParser source code for analyzing OPRA version 2.0 messages is remarkably simple, compact and easy to manage:[1]

```
MATCH "........................................"
MATCH "\x01\x02"
PUSH
MATCH "........."
EXECUTE sequence_number
MATCH "..."
LOOP
  WHILECNT "."
    PUSH
    MATCH "[\x00-\x7f]*[\x80-\xff]"
    EXECUTE action_pmap
    PUSH
(contd ...)
```

```
    WSWITCH
      CASE "[\x80-\xff]"
        EXECUTE action_field
        PUSH
        ENDCASE
      CASE "[\x00-\x7f]"
      MATCH "[\x00-\x7f]*[\x80-\xff]"
        SEND 0x02 0
        EXECUTE action_field
        ENDCASE
    ENDWSWITCH
  ENDWHILECNT
ENDLOOP
```

Intuitively the resulting automaton will start by skipping the IP/UDP headers and will match the OPRA version 2.0 start byte and version number. It will then mark the stream position for the successive action and recognize the initial sequence number for the packet, calling the appropriate user defined code; after that, it will loop and examine every message in the packet to detect first the PMAP and then all individual fields.

User defined actions operate on blocks of input data recognized by the parser and the overall system processing model proceeds along the following steps: (1) look for a section of data –the automaton reads input symbols and switches state until a PUSH action is reached; (2) save the start of data: the current stream position is saved in state machine memory; (3) look for the end of a data field –the automaton reads more input symbols until a state with a user defined action is found; (4) handle the data field –the user defined action is invoked over the block of data from the saved position to the current position.

5 Experimental Results

For performance analysis, we assume that each channel (total 24) can inject messages at full speed by storing the OPRA feeds in main memory, and therefore is not the bottleneck of the decoder. OPRA packets are 400 bytes on average, each containing multiple messages that are encoded using the FAST representation. The messages are typically distributed across the 10-50 byte range, with an average message size of 21 bytes. Thus, each packet contains 19 messages on average. Under the assumption of full injection, we observe that the feeds across the various channels have very similar data pattern and distribution and they tend to have the similar processing rate. Since, the performance is insensitive to the OPRA protocol version, we will consider only OPRA version 2.0 traces in the rest of this paper.

In this section, we present an extensive performance analysis of our algorithm on a variety of multicore architectures. In our tests, we use Intel Xeon Q6600 (Quad), Intel Xeon 5472 (Quad, a.k.a. Harpertown), AMD Opteron 2352 (Quad), IBM Power 6, and Sun UltraSparc T2 (a.k.a. Niagara-2). Power6 and Niagara-2 are hardware multithreaded with 2 threads and 8 threads per core, respectively. Table 1 gives more information about the speeds and feeds of each architecture. We first discuss the performance of the three approaches for processing OPRA FAST messages, the top-down reference implementation, the hand-optimized bottom-up version and DotStar as discussed in sections 2, 3 and 4, respectively. Figure 5a gives the decoding rate per thread in millions of messages per second. We observe

[1] It is worth noting that the full OPRA FAST protocol is described in a 105-page manual!!

that our bottom-up and DotStar implementations are consistently 3 to 4 times faster than the reference implementation; the Intel Xeon E5472 gives the maximum processing rate for a given thread, and that our optimized Bottom-Up and DotStar implementations are similar in terms of performance, with the hand-optimized version that is only 7% faster, on average. The Sun UltraSparc T2 processor is designed to operate with multiple threads per core, thus the single thread performance is much lower than other processors. It is also important to note that our single core OPRA decoding rate is much higher than the current needs of the market, as shown in Figure 1b.

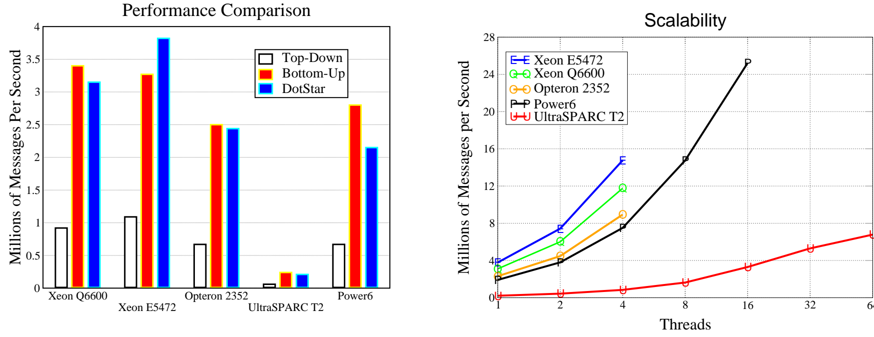

Fig. 5: (a) Performance Comparison, (b) Speedup

	CPU Speed (GHz)	Sockets	Cores /Socket	Threads /Core	Threads	Cache Size (KB)
INTEL Xeon Q6600	2.4	1	4	1	4	4096
INTEL Xeon E5472	3.0	1	4	1	4	6144
AMD Opteron 2352	2.1	1	4	1	4	512
Sun UltraSPARC T2	1.2	1	8	8	64	4096
IBM Power6	4.7	4	2	2	16	4096

Table 1

Figure 5b presents a scalability study of our high-level DotStar implementation. In these tests the threads operate in parallel on different parts of the data from the same OPRA channel. For the Intel Xeon processors our implementation scales almost linearly and processes 15 million messages per second, on a single E5472 quad-core socket and 12 million messages per second, on the Q6600. On Niagara-2, the performance scales linearly up to 16 threads, and reaches 6.8 million messages per second using 64 threads (8 threads/8 cores). For the IBM Power6, we get a scaling advantage up to 16 threads using 8 cores, giving a performance of 26 million messages per second. The performance scales linearly up to 8 threads (1 thread per core), with a performance improvement of 1.5x on 16 threads (2 threads per core).

6 Discussion

In this section we provide detailed insight into the performance results presented in the previous section. The OPRA FAST decoding algorithm can be broken down into 5 main components, processing of the (1) PMAP field, (2) integer fields, (3) string fields, (4) copying previous values, (5) computing index of fields using *clz*. Every OPRA message contains on an average one PMAP field of 5 bytes, integer fields occupying in total 13 bytes, string fields of 4 bytes, in total 22 bytes.

	size	%	INTEL Xeon Q6600 instr	INTEL Xeon Q6600 ns	INTEL Xeon Q6600 cyc	INTEL Xeon E5472 instr	INTEL Xeon E5472 ns	INTEL Xeon E5472 cyc	AMD Opteron 2352 instr	AMD Opteron 2352 ns	AMD Opteron 2352 cyc	SUN UltraSPARC T2 instr	SUN UltraSPARC T2 ns	SUN UltraSPARC T2 cyc	IBM Power6 instr	IBM Power6 ns	IBM Power6 cyc
	1	62.6	3.0	1.4	3.4	3.0	1.5	4.5	3.0	4.0	8.4	3.0	6.5	7.8	4	2.3	10.8
	2	22.1	3.5	1.3	3.2	3.5	1.6	4.9	3.5	4.8	10.1	3.0	6.8	8.2	3.5	2.2	10.3
INT	3	13.6	3.7	1.4	3.4	3.7	1.7	5.2	3.7	5.0	10.5	3.0	5.4	6.5	3.3	2.0	9.4
	4	2.7	3.7	1.5	3.6	3.7	1.8	5.4	3.7	5.1	10.7	3.0	6.6	7.9	3.2	1.9	8.9
	5	0.0	3.8	1.5	3.8	3.8	1.8	5.5	3.8	5.1	10.8	3.0	6.6	7.9	3.2	1.9	8.8
	1	0.0	7.0	2.6	6.3	7.0	3.3	9.9	7.0	9.4	19.7	8.0	17.3	20.7	7	4.4	20.7
	2	0.0	6.0	2.5	6.0	6.0	3.0	9.0	6.0	8.0	16.9	7.4	16.5	19.8	6.5	4.2	19.7
	3	0.0	5.7	2.5	6.0	5.7	2.8	8.5	5.7	7.6	15.9	7.3	16.0	19.2	6.3	4.1	19.0
	4	8.1	5.5	2.4	5.9	5.5	2.8	8.3	5.5	7.4	15.5	7.5	16.9	20.3	6.2	3.9	18.6
PMAP	5	87.1	5.4	2.4	5.8	5.4	2.7	8.1	5.4	7.3	15.4	7.2	15.9	19.1	6.2	3.9	18.1
	6	4.8	5.3	2.3	5.6	5.3	2.7	8.0	5.3	7.1	15.0	7.7	17.5	21.0	6.0	3.8	17.8
	7	0.0	5.2	2.3	5.5	5.2	2.6	7.8	5.2	7.0	14.7	7.9	18.2	21.8	6	3.8	17.9
	8	0.0	5.1	2.3	5.5	5.1	2.6	7.7	5.1	6.9	14.4	7.5	16.9	20.3	6	3.9	18.1
Dcopy/msg			-	11.0	26.4	-	15.0	45.0	-	45.1	94.7	-	706.0	847.2	-	14.0	65.8
Scopy/byte			-	1.4	3.4	-	1.9	5.7	-	4.5	9.5	-	15.2	18.2	-	3.9	18.6
CLZ/field			6	2.8	6.7	5	2.0	6	5	4.1	8.5	17	36.9	44.3	6	4	18.8
			msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b
Basic blocks peak			11.8	3.6	8.5	11.8	3.5	10.6	4.7	9.0	19.0	0.7	59.0	70.8	7.5	5.6	26.2
Actions only			8.8	4.8	11.5	9.5	4.4	20.6	4.3	9.7	20.3	0.6	69.3	83.2	5.3	7.9	37.2
Dotstar rate only			5.8	8.1	19.4	7.5	6.4	30.0	5.7	7.5	15.7	0.4	109.8	131.8	4.1	11.2	52.6
Optimal			3.5	-	-	4.2	-	-	2.5	-	-	0.24	-	-	2.3	-	-
Actions w/Dotstar			**3.1**	13.3	32.0	**3.8**	11.0	51.7	**2.4**	17.5	36.8	**0.2**	200.0	240.0	2.1	20.0	94.0
Optimality ratio			**0.89**	-	-	**0.91**	-	-	**0.96**	-	-	**0.84**	-	-	**0.92**	-	-

Table 2: Optimality analysis

Decoding a message requires copying the last known values into a new data structure, as the encoded message only contains information on a subset of fields. To find the index of each new data fields, we perform a *clz* (count leading zeroes) operation. For each of these actions we calculate the instructions from assembly level, Table 2 gives the instructions per byte for processing the PMAP, integer and string fields, on each multicore processor. For nsec and cycles per byte we compute actual performance, by performing multiple iterations of the algorithm with and without the corresponding action and normalizing the difference. Similarly, for the *clz* action, we compute cycles, nsec, and instructions per data field, and for the *copy msg* we compute average nsec and cycles/message.

Using the field length distributions, their occurrence probability rates, and individual action performance, we compute the aggregate estimated peak performance in the first row of the last block in the table. Note that this peak estimate does not take into account any control flow or I/O time. In the second row we give the actual actions-only performance that is computed by taking the difference between the total performance and performance after commenting out the actions, lets call it A. The DotStar rate is the peak performance of our parsing algorithm on OPRA feeds, that does not include any actions on the recognized data fields, lets call it B. We notice that: The *optimal* row in that block is the estimated peak performance that we should have got by combining the DotStar routine and our optimized kernels, that is computed using the formula $(\frac{1}{A} + \frac{1}{B})^{-1}$. We compare this to our actual performance and compute the *optimality ratio* in the last row of the table. We observe the following.

- On the IBM Power6, we get similar nsec/byte performance as compared to Intel Xeon processor. We believe that our code is not able to take advantage of the deep pipeline and in-order execution model of the Power6.
- To get maximum performance for the *copy msg* routine, we developed our own vectorized memory copy using SSE intrinsics, and pipelined load and stores to obtain best performance. On the Sun Niagara-2 we used a manual copy, at the granularity of an integer, and unrolled the load and store instructions. With vectorized memory copy we get about 2x performance advantage over the *memcpy* library routine, whereas for the manual scalar copy we get an advantage of 1.2x on Sun Niagara-2.
- On Sun-Niagara, the *copy* action is much slower than on other processors.
- Our implementation requires integer/string processing as opposed to floating point computation, this fails to utilize the much improved floating point units on Sun Niagara-2.
- On Intel Q6600, cycles/byte performance matches closely with instructions/byte count.
- Our implementation is close to the optimal performance obtained with the high-level approach, with an average optimality ratio of 0.9.
- Table 3 gives the latency for decoding/processing an OPRA message on each of the multicore processors discussed in the papers. We decode one message at a time on a single processing thread, thus the average latency to process a message is given by taking the inverse of the processing rate. We observe a latency between 200 and 500 nsec on the Intel/IBM/AMD processors, which accounts for only a negligible latency inside the ticker plant, under a very high throughput.

7 Conclusions

The increasing rate of market data traffic is posing a very serious challenge to the financial industry and to the current capacity of trading systems worldwide. This requires solutions that protect the inherent nature of the business model, i.e., providing low processing latency with ever increasing capacity requirements. We presented a novel solution for OPRA

	INTEL Xeon Q6600	INTEL Xeon E5472	AMD Opteron 2352	Sun UltraSparc T2	IBM Power6
Latency/msg (nsec)	317	261	409	4545	476

Table 3

FAST feed decoding and normalization, on commodity multicore processors. Our approach captures the essence of OPRA protocol specification in a handful of lines of DSParser, the high-level descriptive language that is the programming interface of the DotStar protocol parser, thus providing a solution that is high-performance, yet flexible and adaptive. We showed impressive processing rates of 15 million messages per second on the fastest single socket Intel Xeon, and over 24 million messages per second using the IBM Power6 on a server with 4 sockets. We presented an extensive performance evaluation to understand the intricacies of the decoding algorithm, and exposed many distinct features of the multicore processors, that can be used to estimate performance. In the future, we plan to extend our approach to other components of the ticket plant, and evaluate a threaded-model in DotStar, to help pipeline independent analytics processing kernels seamlessly.

References

1. Virat Agarwal, Lurng-Kuo Liu, and David Bader. Financial Modeling on the Cell Broadband Engine. *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, April 2008.
2. V.M. Glushkov. The abstract theory of automata. *Russian Mathematical Survey. v16. 1-53*.
3. Mike Kistler, Michael Perrone, and Fabrizio Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, 25(3), May/June 2006.
4. Dinesh Manocha. General-Purpose Computations Using Graphics Procecssors. *IEEE Computer*, 38(8):85–88, Aug. 2005.
5. Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of Decision Problems for Simple Regular Expressions. In *MFCS*, pages 889–900, 2004.
6. Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, New York, NY, USA, 2002.
7. OPRA. Option Price Reporting Authority. Available at `http://www.opradata.com/`.
8. P. Overell. Augmented BNF for Syntax Specifications: ABNF, 1997.
9. Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *IMC'06: Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 289–300, New York, NY, USA, 2006. ACM.
10. Davide Pasetto and Fabrizio Petrini. DotStar: Breaking the Scalability and Performance Barriers in Regular Expression Set Matching. Technical report, IBM T.J. Watson, 2009.
11. Srinivasan R. RPC: Remote Procedure Call Protocol Specification Version 2, 1995.
12. TABB Group. Trading At Light Speed: Analyzing Low Latency Data Market Data Infrastructure. Available at `http://www.tabbgroup.com/`, 2007.