Towards Highly Parallel Event Processing through Reconfigurable Hardware

Mohammad Sadoghi, Harsh Singh, Hans-Arno Jacobsen
Middleware Systems Research Group
Department of Electrical and Computer Engineering
University of Toronto, Canada

ABSTRACT

We present *fpga-ToPSS* (Toronto Publish/Subscribe System), an efficient event processing platform to support high-frequency and low-latency event matching. *fpga-ToPSS* is built over reconfigurable hardware—FPGAs—to achieve line-rate processing by exploring various degrees of parallelism. Furthermore, each of our proposed FPGA-based designs is geared towards a unique application requirement, such as flexibility, adaptability, scalability, or pure performance, such that each solution is specifically optimized to attain a high level of parallelism. Therefore, each solution is formulated as a design trade-off between the degree of parallelism versus the desired application requirement. Moreover, our event processing engine supports Boolean expression matching with an expressive predicate language applicable to a wide range of applications including real-time data analysis, algorithmic trading, targeted advertisement, and (complex) event processing.

1. INTRODUCTION

Efficient event processing is an integral part of growing number of data management technologies such as real-time data analysis [27, 5, 29], algorithmic trading [26], intrusion detection system [5, 8], location-based services [30], targeted advertisements [9, 25], and (complex) event processing [1, 7, 2, 6, 17, 3, 24, 25, 9].

A prominent application for event processing is algorithmic trading; a computer-based approach to execute buy and sell orders on financial instruments such as securities. Financial brokers exercise investment strategies (*subscriptions*) using autonomous high-frequency algorithmic trading fueled by real-time market *events*. Algorithmic trading is dominating financial markets and now accounts for over 70% of all trading in equities [11]. Therefore, as the computer-based trading race among major brokerage firms continues, it is crucial to optimize execution of buy or sell orders at the microsecond level in response to market events, such as corporate news, recent stock price patterns, and fluctuations in currency exchange rates, because every microsecond translates into opportunities and ultimately profit [11]. For instance, a simple classical arbitrage strategy has an estimated annual profit of over $21 billion according to TABB Group [12]. Moreover, every 1-millisecond reduction in response-time is estimated to generate the staggering amount of over $100 million a year [19]; such requirements greatly increases the burden placed on event processing platform.

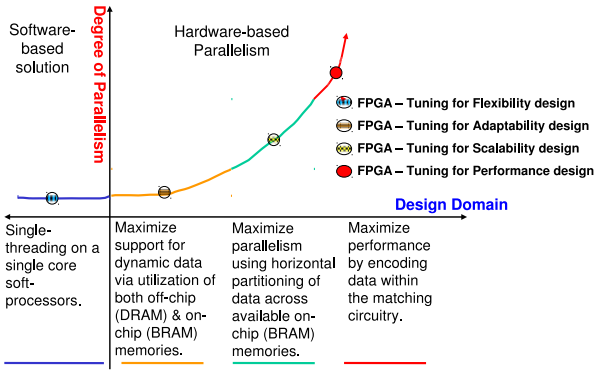

Figure 1: Degrees of Offered Parallelism

Therefore, a scalable event processing platform must efficiently determine all subscriptions that match incoming events at a high rate, potentially up to a million events per second [4]. Similar requirements are reported for event processing in network monitoring services [27].

To achieve throughput at this scale, we propose and evaluate a number of novel FPGA-based event processing designs (Field Programmable Gate Array). An FPGA is an integrated circuit designed to be reconfigurable to support custom-built applications in hardware. Potential application-level parallelism can be directly mapped to purpose-built processing units operating in parallel. Configuration is done through encoding the application in a programming language-style description language and synthesising a configuration uploaded on the FPGA chip [14]. FPGA-based solutions are increasingly being explored for data management tasks [20, 21, 22, 28, 26].

This promising outlook has a few caveats that make the acceleration of any data processing with FPGAs a challenging undertaking. First, current FPGAs (e.g., 800MHz Xilinx Virtex 6) are still much slower compared to commodity CPUs (e.g. 3.2 GHz Intel Core i7). Second, the accelerated application functionality has to be amenable to parallel processing. Third, the on-/off-chip data rates must keep up with chip processing speeds to realize a speedup by keeping the custom-built processing pipeline busy. Finally, FPGAs restrict the designer's flexibility and the application's dynamism[1], both of which are hardly a concern in standard software solutions. However, the true success of FPGAs is rooted in three distinctive features: hardware parallelism, hardware reconfigurability, and substantially higher throughput rates.

[1]e.g., subscription insert and delete operations are not a given.

DaMoN'11, June 13, 2011, Athens, Greece.

Thus, each of our solutions is formulated as a design trade-off between the degree of exploitable parallelism (cf. Fig. 1) versus the desired application-level requirements. Requirements considered are: the ease of the development and deployment cycle (*flexibility*), the ability of updating a large subscription workload in real-time (*adaptability*), the power of obtaining a remarkable degree of parallelism through horizontal data partitioning on a moderately sized subscription workload (*scalability*), and, finally, the power of achieving the highest level of throughput by eliminating the use of memory and by specialized encoding of subscriptions on FGPA (*performance*).

We experiment with four novel system designs that exhibit different degrees of parallelism (cf. Fig. 1) and capture different application requirements. In our application context, achievable performance is driven by the degree of parallelism (in which FPGAs dominate) and the chip operating frequency (in which CPUs dominate). Therefore, our solution design space is as follows: a single thread running on single CPU core (PC), a single thread on a single soft-processor (*flexibility*) [2], up to four custom hardware matching units (MUs) running in parallel in which the limiting factor is off-chip memory bandwidth (*adaptability*), horizontally partitioning data across m matching units running in parallel in which the limiting factor is the chip resources and the on-chip memory (*scalability*), and, lastly, n (where $n \geq m$) matching units running in parallel (with no memory access because the data is also encoded on the chip), in which the limiting factor is the amount of chip resources, particularly, the required amount of wires (*performance*).

The ability of an FPGA to be re-configured on-demand into a custom hardware circuit with a high degree of parallelism is key to its advantage over commodity CPUs for data and event processing. Using a powerful multi-core CPU system does not necessarily increase processing rate (Amdahl's Law) as it increases inter-processor signaling and message passing overhead, often requiring complex concurrency management techniques at the program and OS level. In contrast, FPGAs allow us to get around these limitations due to their intrinsic highly inter-connected architecture and the ability to create custom logic on the fly to perform parallel tasks. In our design, we exploit parallelism, owing to the nature of the matching algorithm (Sec. 3), by creating multiple matching units which work in parallel with multi-giga bit throughput rates (Sec. 4), and we utilize reconfigurability by seamlessly adapting relevant components as subscriptions evolve (Sec. 4).

2. RELATED WORK

FPGA An FPGA is a semiconductor device with programmable lookup-tables (*LUTs*) that are used to implement truth tables for logic circuits with a small number of inputs (on the order of 4 to 6 typically). FPGAs may also contain memory in the form of flip-flops and block RAMs (BRAMs), which are small memories (a few kilobits), that together provide small storage capacity but a large bandwidth for circuits in the FPGA. Thousands of these building blocks are connected with a programmable interconnect to implement larger-scale circuits.

Past work has shown that FPGAs are a viable solution for building custom accelerated components [20, 21, 22, 28, 26]. For instance, [20] demonstrates a design for accelerated XML processing, and [21] shows an FPGA solution for processing market feed data. As opposed to these approaches, our work concentrates on supporting a more general event processing platform specifically designed to accelerate the event matching computation. Similarly, [22] presents a data stream processing framework that uses FPGAs to efficiently run hardware-encoded queries (without join). Lastly, on the path toward supporting stream processing, a novel framework on how to efficiently run hardware-encoded regular expression queries in parallel on an FPGA is proposed [28]; a key insight was the realization that the deterministic finite automata (DFA), although suitable for software solutions, results in explosion of space; thereby, to bound the required space on the FPGA, a non-deterministic finite automata (NFA) is utilized. Our approach differs from [28] as we primarily focus on supporting large scale conjunctive Boolean queries.

On a different front, a recent body of work has emerged that investigates the use of new hardware architectures for data management systems [10]; for instance, multi-core architectures were utilized to improve the performance of database storage manager [13] and to enhance the transaction and query processing [23].

Finally, the sketch of our initial proposal was presented in [26]; our current work not only delves into much more technical depth which was omitted from [26], it also reformulates our FPGA-based solutions based on the extent of parallelism in each design, a formulation that permits a concrete performance comparison of various designs accompanied by a comprehensive experimental analysis. Most importantly, this work introduces key insights and novel system designs through introducing an effective horizontal data partitioning to achieve an unprecedented degrees of parallelism on moderate-size subscription workload.

Matching The matching is one of the main computation intensive components of event processing which has been well studied over the past decade (e.g., [1, 7, 2, 6, 17, 3, 9, 24, 25]). In general, the matching algorithms are classified as (1) counting-based [7], and (2) tree-based [1, 25]. The counting algorithm is based on the observation that subscriptions tend to share many common predicates; thus, the counting method minimizes the number of predicate evaluations by constructing an inverted index over all unique predicates. Similarly, the tree-based methods are designed to reduce predicate evaluations; in addition, they recursively cut through space and eliminate subscriptions on the first encounter with an unsatisfiable predicate. The counting- and tree-based approaches can be further classified as either key-based (in which for each subscription a set of predicates are chosen as identifiers [7]), or as non-key method [1]. In general, the key-based methods reduce memory access, improve memory locality, and increase parallelism, which are essentials for a hardware implementation. One of the most prominent counting-based matching algorithms are Propagation [7], a key-based method while one of the most prominent tree-based approach, BE-Tree, which is also a key-based method [25].

3. EVENT PROCESSING MODEL

Subscription Language & Semantics The matching algorithm takes as input an event (e.g., market event and user profile) and a set of subscriptions (e.g., investment strategies and targeted advertisement constraints) and returns matching subscriptions. The event is modeled as a value assignment to attributes and the subscription is modeled as a Boolean expression (i.e., as conjunction of Boolean predicates). Each Boolean predicate is a triple of either [attribute$_i$, operator, values] or [attribute$_i$, operator, attribute$_j$]. Formally, the matching problem is defined as follows: *given an event* e *and a set of subscriptions* $\mathbf{s}$*, find all subscriptions* $s_i \in \mathbf{s}$ *satisfied by* e.

Matching Algorithm The Propagation algorithm is a state-of-the-art key-based counting method that operates as follows [7]. First, each subscriptions is assigned a key (a set of predicates) based on which the typical counting-based inverted index is replaced by a

[2] A soft-processor is a processor encoded like an application running on the FPGA. It supports compiled code written in a higher-level language, like for example C without operating system overhead.

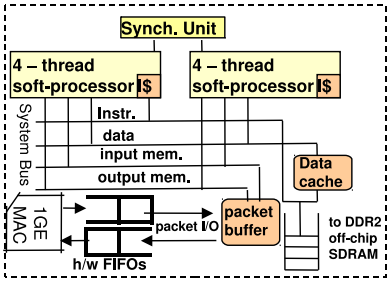

Figure 2: Tunning for Flexibility Design

set of multi-attribute hashing schemes. The multi-attribute hashing scheme uniquely assigns subscriptions into a set of disjoint clusters. Second, keys are selected from a candidate pool using a novel cost-based optimization tuned by the workload distribution to minimize the matching cost [7]. The Propagation data structure has three main strengths which makes it an ideal candidate for a hardware-based implementation: (1) subscriptions are distributed into a set of disjoint clusters which enables highly parallelizable event matching through many specialized custom hardware matching units (MUs), (2) within each cluster, subscriptions are stored as contiguous blocks of memory which enables fast sequential access and improves memory locality, and (3) the subscriptions are arranged according to their number of predicates which enables prefetching and reduces memory accesses and cache misses [7].

4. FPGA-BASED EVENT PROCESSING

Commodity servers are not quite capable of processing event data at line-rate. The alternative is to acquire and maintain high cost purpose-built event processing applications. In contrast, our design uses an FPGA to significantly speed up event processing computations involving event matching. FPGAs offer a cost effective event processing solutions, since custom hardware can be altered and scaled to adapt to the prevailing load and throughput demands. Hardware reconfigurability allows FPGAs to house *soft-processors*—processors composed of programmable logic. A soft-processor has several advantages: it is easier to program on it (e.g., using `C` as opposed to `Verilog` which requires specialized knowledge and hardware development tools), it is portable to different FPGAs, it can be customized, and it can be used to communicate with other components and accelerators in the design. In this project, the FPGA resides on a NetFPGA [18] network interface card and communicates through DMA on a PCI interface to a host computer. FPGAs have programmable I/O pins that in our case provide a direct connection to memory banks and to the network interfaces, which in a typical server, are only accessible through a network interface card.

In this section, we describe our four implemented designs each of which is optimized for a particular characteristic such as flexibility in development and deployment process, adaptability in supporting changes for a large workload size, scalability through horizontal data partitioning for moderate workload size, and performance in maximizing throughput for small workload size. Most notably, the distinguishing feature among our proposed designs is the level of parallelism that ranges from running all subscriptions on a single processor (*flexibility*) to running every subscription on its own custom hardware unit (*performance*).

4.1 Tuning for Flexibility

Our first approach is the soft-processor(s)-based solution (cf. Fig. 2), which runs on a soft-processor that is implemented on the NetFPGA platform. This solution also runs the same C-based event matching code that is run on the PC-based version (our baseline);

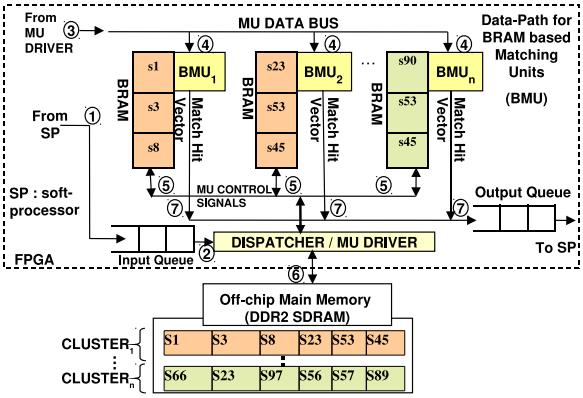

Figure 3: Tuning for Adaptability Design

thus, this design is the easiest to evolve as message formats and protocols change. In order to maximize throughput of our event processing application, we chose NetThreads [15] as the baseline soft-processor platform for the FPGA. NetThreads has two single-issue, in-order, 5-stage, 4-way multi-threaded processors (cf. Fig. 2), shown to deliver more throughput than simpler soft-processors [16]. In a single core, instructions from four hardware threads are issued in a round-robin fashion to hide stalls in the processor pipeline and execute computations even when waiting for memory. Such a soft-processor system is particularly well-suited for event processing: The soft-processors suffer no operating system overhead compared to conventional computers, they can receive and process packets in parallel with minimal computation cost, and they have access to a high-resolution system clock (much higher than a PC) to manage timeouts and scheduling operations. One benefit of not having an operating system in NetThreads is that packets appear as character buffers in a low latency memory and are available immediately after being fully received by the soft-processor (rather than being copied to a user-space application). Also, editing the source and destination IP addresses only requires changing a few memory locations, rather than having to comply with the operating system's internal routing mechanisms. Because a simpler soft-processor usually executes one instruction per cycle, it suffers from a raw performance drawback compared to custom logic circuits on FPGAs; a custom circuit can execute many operations in parallel as discussed next.

4.2 Tuning for Adaptability

In order to utilize both hardware-acceleration while supporting large dynamic subscriptions on both off-chip and on-chip memories, we propose a second scheme (cf. Fig. 3). Since FPGAs are normally programmed in a low-level hardware-description language, it would be complex to support a flexible communication protocol. Instead, we instantiate a soft-processor (SP) to implement the packet handling in software. After parsing incoming event data packets, the soft-processor offloads the bulk of the event matching to a dedicated custom hardware matching unit. Unlike the subscription-encoded matching units used in the tuned for *performance* design, these matching units use low-latency on-chip memories, Block RAMs (BRAMs) available on FPGAs, that can be stitched together to form larger dedicated blocks of memory. The FPGA on the NetFPGA platform [18] has 232 18kbit BRAMs which are partially utilized to cache a subset of subscriptions. Having an on-chip subscription data cache allows event matching to be initiated even before the off-chip subscription data can be accessed. However, our matching algorithm leverages data locality in the storage of dynamic subscriptions, which may be updated during run time, in contiguous array clusters thereby exploiting burst-oriented

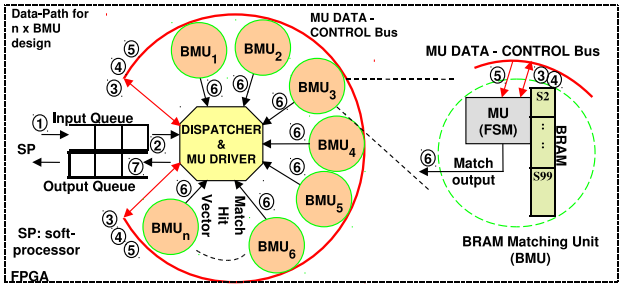

Figure 4: Tuning for Scalability Design

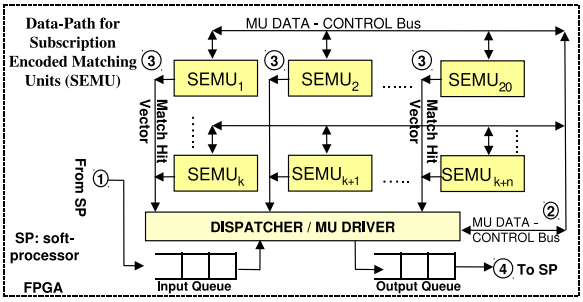

Figure 5: Tuning for Performance Design

data access feature of the DDR2 (or SDRAM), off-chip memory, while fetching the subscription data clusters. Thus, having an on-chip subscription cache partially masks the performance penalty (latency) of fetching the subscriptions from the off-chip DDR2 memory, which is the main throughput bottleneck of our FPGA-based event processing solution. Therefore, the maximum amount of useful parallelism in this design is limited by the memory bandwidth; in particular, no more than four custom hardware matching units can be sustained simultaneously; any additional matching units will remain idle because only a limited amount of data can be transfered from off-chip memory to on-chip memory in each clock cycle.

In our design tuned for *adaptability*, we employ a more generalized design that enables the matching units to support a dynamic and a larger subscription workload than can be supported in our designs that tuned either for *scalability* or *performance*. Our *adaptability* design employs the BRAM-based Matching Units (BMUs) which allows a subset of subscriptions to be stored on the on-chip dedicated low latency BRAMs; thus making the design less hardware resource intensive compared to the our pure hardware implementation (tuned for *performance* design). Furthermore, coalescing dynamic subscription data into an off-chip memory image is achieved using the Propagation algorithm. The resulting subscription data image is downloaded to the off-chip main memory (e.g DDR2 SDRAM) while loading FPGA configuration bitstream. Nevertheless, any hardware performance advantage promised by a FPGA-based design soon dwindles when the data must be accessed from an off-chip memory. We adopt two approaches to reduce the impact of off-chip memory data access latency on the overall system throughput. Firstly, we take advantage of high degree of the data locality inherent in Propagation's data structure which helps to minimize random access latency. Secondly, to achieve locality subscriptions are grouped into non-overlapping clusters using attribute-value pair as access keys. Therefore, this data structure is optimized for storing large number of subscriptions in off-chip memory. In addition, we incorporate a fast (single cycle latency) but smaller capacity BRAMs for each matching unit to store subset of subscriptions, which helps mask the initial handshaking setup delay associated with off-chip main memory access, i.e., the event matching can begin against these subscriptions as soon as the event arrives; in the meantime the system prepares to setup data access from the off-chip DDR2 main memory.

The stepwise operation of this design is depicted in Fig. 3. Upon arrival of an event, the SP transfers (1) the data packets to the input queue of the system. A custom hardware submodule, the DISPATCHER unit, extracts subscription predicates-value pairs, which are input to hash functions to generate cluster addresses. Cluster addresses are used to look-up the memory locations (2) of the relevant subscription clusters residing both in BMU BRAMs and in off-chip main memory. The DISPATCHER then feeds the event (3) and previously computed cluster addresses (4) on the MU DATA BUS (common to all BMUs). Next, the MU DRIVER unit activates all parallel BMUs to initiate matching (5) using on-chip static subscriptions stored in each BMU, while simultaneously queuing up read requests for the off-chip main memory. The transfer (6) of dynamic subscription data between the BMUs is pipelined to avoid stalling the matching units due to data exhaustion. Finally the match results are pushed (7) into the output queue from which the SP transfers the results to the network interface to be sent to the intended host(s).

4.3 Tuning for Scalability

The key property of our proposed design tuned for *scalability* is the horizontal data partitioning that maximizes parallelism (cf. Fig. 4). This design offers the ability to adjust the required level of parallelism (which directly translates into matching throughput) by adjusting the degree of data partitioning for a moderate size workload, yet without significantly compromising the feature offered in our *adaptability* design. It achieves this by fully leveraging the available on-chip (BRAM) memory to partition the global Propagation's data structure across BRAM blocks such that each subset of BRAMs is dedicated to each matching unit, in which the matching unit has an exclusive access to a chunk of the global Propagation's structure. Unlike our *adaptability* design in which the degree of parallelism is quite restricted due to the off-chip memory's access latency, resulting in several data starved or stalled matching units, this design employs matching units (BMUs) (cf. Fig. 4) that are each provisioned with a dedicated BRAM memory in order to keep them fully supplied with subscription data. Therefore, the degree of parallelism achieved is simply a function of the number of BMUs that can be supported by the underlying FPGA. Finally, a non-performance critical soft-processor (SP) can be employed to update the on-chip memory tables attached to each BMU in the design; hence, supporting dynamic subscription workload.

The overall stepwise operation of our tuned for *scalability* design, depicted in Fig. 4, is similar to that which occurs in the tuned for *adaptability* design for steps (1) to (5), with the difference being in the absence of the off-chip main memory used for storing the dynamic subscriptions. The operation and logic of the DISPATCHER and MU DRIVER submodule is further simplified as the off-chip memory access arbitration and data dissemination to BMUs is eliminated. Every BMU consists of a four-state Finite State Machine, that upon receiving the event data (3) initiates matching by sequentially fetching one subscriptions every clock cycle from the dedicated BRAM memory containing the cluster starting at the address (4) that was dispensed by the DISPATCHER unit. Since all BMUs are ran in parallel and in sync with each other, the DISPATCHER must dispense the next cluster address only when all BMUs have completed matching all subscriptions in the current cluster. In final phase (6), once all BMUs finish matching all the subscriptions' clusters corresponding to the predicates present in the incoming event, the final result tallying phase is initiated where matched subscriptions or number of matches found are placed on the match hit

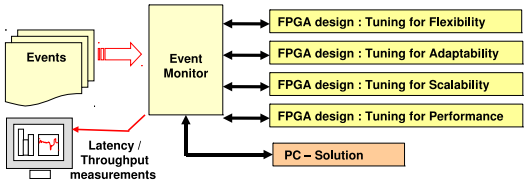

Figure 6: Evaluation Testbed

vectors and consolidated as a final result value by the DISPATCHER unit to be transfered to SP via the output queue.

4.4 Tuning for Performance

Our final approach (cf. Fig. 5) is a purely hardware solution: custom hardware components perform necessary steps involving event parsing and matching of event data against subscriptions. This method provides near line-rate performance, but also involves a higher level of complexity in integrating custom heterogeneous accelerators in which both the performance-critical portion of the event processing algorithm and the encoding of subscriptions are incorporated within the design of the matching unit logic; thereby, completely eliminating all on- and off-chip memory access latencies. Essentially, each subscription is transformed into a self-contained custom hardware unit; this subscription encoding achieves the highest level of parallelism in our design space because all subscriptions are ran in parallel.

The *performance* design offers the highest rate at which incoming events can be matched against subscriptions, which are encoded in the SUBSCRIPTION ENCODED MATCHING UNIT (SEMU) logic on the FPGA. This method avoids the latency of both on and off-chip memory access, but significantly constraints the size of the subscription base that can be supported. A diagram of this design is shown in Fig. 5. This setup is massively parallelized and offers event matching at extremely high rates (i.e. one per clock cycle).

The stepwise operation of the our tuned for *performance* design is depicted in Fig. 5. In this design, the soft-processor (SP) only serves to transfer (1) the received event data packets from the network interface input buffer to the input queue of the our system. Custom hardware submodule, the DISPATCHER module, parses (2) the incoming events and feeds the current event data to all the matching units while the MU DRIVER module generates all the necessary control signals to run all SEMUs synchronously. Each unit is able to match all encoded subscriptions against the current event in one clock cycle. However, subsequent clock cycles are spent in tallying the matches and preparing the final responses (e.g. forward address look-up or consolidating system wide match counts) that is eventually pushed (3) into the output queue. The SP then transfers (4) the final result from the output queue to the network interface to be sent to the intended host(s).

5. EXPERIMENTAL RESULTS

This section describes our evaluation setup including the hardware used to implement our FPGA-based event processing system and the measurement infrastructure.

Evaluation Platform Our FPGA based solutions are instantiated on the NetFPGA 2.1 [18] platform, operating at 125MHz and have access to four 1GigE Media Access Controllers (MACs) via high-speed hardware FIFO queues (cf. Fig. 2) allowing a theoretical 8Gbps of concurrently incoming and outgoing traffic capacity. In addition, a memory controller to access the 64 Mbytes of on-board off-chip DDR2 SDRAM is added. The system is synthesized to meet timing constraints with the Xilinx ISE 10.1.03 tool and targets a Virtex II Pro 50 (speed grade 7ns). Our soft-processor and matching units run at the frequency of the Ethernet MACs (125MHz).

	1x MU	4x MUs	32x MUs	128x MUs
250	7.5	5.5	5.0	3.6
1K	9.3	6.1	4.3	4.3
10K	64.0	19.0	6.8	5.4
50K	223.5	59.9	12.3	7.3

Table 1: Latency (μs) vs. the # of MUs (Scalability Design)

Evaluation & Evaluation Setup For our experiments, we used HP DL320 G5p servers (Quad Xeon 2.13GHz) equipped with an HP NC326i PCIe dual-port gigabit network card running Linux 2.6.26. As shown in Fig. 6, we exercised our event processing solutions from the server executing a modified `Tcpreplay 3.4.0` that sends event packet traces at a programmable fixed rate. Packets are timestamped and routed to either the FPGA-based designs or PC-based design. Each FPGA-based design is configured as one of the solutions described in Sec. 4 and PC-based is a baseline serving as comparison only. The network propagation delays are similar for all designs. Both FPGA-based or PC-based designs forward market events on the same wire as incoming packets which allows the Event Monitor (EM), cf. Fig. 6, to capture both incoming and outgoing packets from these designs. The EM provides a $8ns$ resolution on timestamps and exclusively serves for the measurements.

Evaluation Workload We generate a workload of tens of thousands of subscriptions derived from investment strategies such as arbitrage and buy-and-hold. In particular, we vary the workload size from 250 subscriptions to over 100K subscriptions. In addition, we generate market events using the Financial Information eXchange (FIX) Protocol with FAST encoding[3].

Evaluation Measurements We characterize the system throughput as the maximum sustainable input packet rate obtained through a bisection search: the smallest fixed packet inter-arrival time where the system drops no packets when monitored for five seconds—a duration empirically found long enough to predict the absence of future packet drops at the given input rate. The latency of our solutions is the interval between the time an event packet leaves the Event Monitor output queue to the time the first forwarded version of the market event is received and is added to the output queue of the Event Monitor.

5.1 FPGA Performance Benefits

Packet Processing Measuring the baseline packet processing latency of both PC and FPGA-based solutions is essential in order to establish a basis for comparison. When processing packets using the PC solution, we measured an average round-trip time of $49\mu s$ with a standard deviation of $17\mu s$. With the NetThreads processor on the FPGA replying, we measure a latency of $5\mu s$ with a standard deviation of $44ns$. Because of the lack of operating system and more deterministic execution, the FPGA-based solution provides a much better bound on the expected packet processing latency; hence, our FPGA-based solution outperformed the PC-based solution in baseline packet processing by orders of magnitude.

Event Processing Before we begin our detailed comparison of various designs, we study the effect of the number of matching units (MUs) on the matching latency for our *scalability* design, Table 1. As expected, as we increase the number of MUs, moving from 1 MU to 128 MUs, the latency is improved significantly especially for the larger subscriptions workload (with chip resources permitting). This improvement is directly proportional to the degree of parallelism obtained by using a larger number of MUs.

In Table 2, we demonstrate the system latency as the subscription workload size changes from 250 to 100K. In summary, even though our FPGA (125MHz Virtex II) is much slower than the latest FPGA

[3] `fixprotocol.org`

	PC	Flexibility	Adaptability	Scalability	Performance
250	53.9	71.0	6.4	3.6	3.2
1K	60.7	199.4	7.5	4.3	N/A
10K	150.0	1,617.8	87.8	5.4	N/A
100K	2,001.2	16,422.8	1,307.3	N/A	N/A

Table 2: End-to-end System Latency (μs)

	PC	Flexibility	Adaptability	Scalability	Performance
250	122,654	14,671	282,142	740,740	1,024,590
1K	66760	5,089	202,500	487,804	N/A
10K	9594	619	11,779	317,460	N/A
100K	511	60	766	N/A	N/A

Table 3: System Throughput (market events/sec)

(800MHz Virtex 6) and significantly slower than our CPU (Quad Xeon 2.13GHz), our deign tuned for *adaptability* is 8x faster than the PC-based solution on workload sizes of less than 1K and continued to improve over the PC solution by up to a factor of two on workload of sizes of 100K. Similarly, the design tuned for *performance*, while currently feasible only for smaller workloads due to lack of resources on the FPGA, is 21.2x faster. Most importantly, our design tuned for *scalability* takes advantage of both of our *adaptability* and *performance* designs by finding the right balance between using the on-chip memory to scale the workload size while using the highly parallel nature of the *performance* design to scale the event processing power. Thus, the *scalability* design is 16.2x faster than our *adaptability* design and is 27.8x faster than the PC design. In addition, a similar trend was also observed for the system throughput experiment as shown in Table 3.

Therefore, the *adaptability* design is limited because of slower off-chip memory bandwidth which greatly hinders the degree of parallelism while the *performance* design is limited because encoding the subscriptions in the logic fabric of the chip consumes much more area than storing them in BRAM or DDR2 providing much denser storage. Finally, contrary to general perspective that software solution cannot be utilized in hardware, the success of our *scalability* design (which adapts a software-based solution) suggests that in order to scale our solution to large subscription workloads, certain software data structures for data placement become a viable solution in conjunction with hardware acceleration and parallelism.

6. CONCLUSIONS & DISCUSSIONS

We observe that event processing is at the core of many data management applications such as real-time network analysis and algorithmic trading. Furthermore, to enable the high-frequency and low-latency requirements of these applications, we presented an efficient event processing platform over reconfigurable hardware that exploits the high degrees of hardware parallelism for achieving line-rate processing. In brief, the success of our *fpga-ToPSS* framework is through the use of reconfigurable hardware (i.e., FPGAs) that enables hardware acceleration using custom logic circuits and elimination of OS layer latency through on-board event processing together with hardware parallelism and novel horizontal data partitioning scheme. As a result, our design tuned for *performance* outperformed the PC-based solution by a factor of 27x on small size subscription sets while our design tuned for *scalability* outperformed the PC-based solution by a factor of 16x even as the workload size was increased; in fact, this gap further widens as workload size increases due an increased opportunity to process a larger amount of data in parallel.

7. REFERENCES

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC'99*.

[2] G. Ashayer, H. K. Y. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. ICDCSW'02.

[3] L. Brenna, A. Demers, J. Gehrke, M. Hong, Ossher, Panda, Riedewald, Thatte, and White. Cayuga: high-performance event processing engine. *SIGMOD'07*.

[4] J. Corrigan. Updated traffic projections. *OPRA, March'07*.

[5] C. Cranor, T. Johnson, and O. Spataschek. Gigascope: a stream database for network applications. In *SIGMOD'03*.

[6] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE'02*.

[7] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for fast pub/sub systems. *SIGMOD'01*.

[8] A. Farroukh, M. Sadoghi, and H.-A. Jacobsen. Towards vulnerability-based intrusion detection with event processing. In *DEBS'11*.

[9] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. In *SIGMOD'10*.

[10] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD'08*.

[11] K. Heires. Budgeting for latency: If I shave a microsecond, will I see a 10x profit? *Securities Industry, 1/11/10*.

[12] R. Iati. The real story of trading software espionage. *TABB Group Perspective, 10/07/09*.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT'09*.

[14] I. Kuon, R. Tessier, and J. Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.'08*.

[15] M. Labrecque et al. NetThreads: Programming NetFPGA with threaded software. In *NetFPGA Dev. Workshop'09*.

[16] M. Labrecque and J. G. Steffan. Improving pipelined soft processors with multithreading. In *FPL'07*.

[17] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. ICDCS '05.

[18] J. W. Lockwood et al. NetFPGA - an open platform for gigabit-rate network switching and routing. In *MSE'07*.

[19] R. Martin. Wall street's quest to process data at the speed of light. *Information Week, 4/21/07*.

[20] A. Mitra et al. Boosting XML filtering with a scalable FPGA-based architecture. *CIDR'09*.

[21] G. W. Morris et al. FPGA accelerated low-latency market data feed processing. *IEEE 17th HPI'09*.

[22] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *VLDB'09*.

[23] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB'10*.

[24] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. GPX-Matcher: a generic boolean predicate-based XPath expression matcher. In *EDBT'11*.

[25] M. Sadoghi and H.-A. Jacobsen. BE-Tree: An index structure to efficiently match boolean expressions over high-dimensional discrete space. In *SIGMOD'11*.

[26] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. In *VLDB '10*.

[27] D. Srivastava, L. Golab, R. Greer, T. Johnson, J. Seidel, V. Shkapenyuk, O. Spatscheck, and J. Yates. Enabling real time data analysis. *PVLDB'10*.

[28] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *PVLDB'10*.

[29] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD'06*.

[30] Z. Xu and H.-A. Jacobsen. Processing proximity relations in road networks. SIGMOD '10.