# Accelerating Publish/Subscribe Matching on Reconfigurable Supercomputing Platform

K.H. Tsoi, I. Papagiannis, M. Migliavacca, W. Luk and P. Pietzuch

Department of Computing, Imperial College London

*Abstract*—**A modular design is proposed and analyzed for accelerating the publish/subscribe matching algorithm in reconfigurable hardware. With help from a performance model, we demonstrate an optimized FPGA implementation which is scalable and efficient enough for many of today's most demanding web and financial applications. Our design achieves 5.9 times speedup over software while consuming around $0.5\%$ of power.**

## I. INTRODUCTION

Today millions of users are subscribing to content over a network by setting sets of filtering rules. Web applications, such as Twitter and Facebook, enable users to specify their interests and subscribe to updates provided by information publishers. This publish/subscribe (pub/sub) model for content distribution can also be found in financial data processing and monitoring applications [1]. In general, a stream of information updates in the form of event messages (or events), must be matched against a set of defined rules (or subscriptions). The goal of pub/sub matching is to notify a potentially large set of subscribers about matching events quickly (in terms of latency) and efficiently (in terms of throughput).

This matching problem has become a performance bottleneck as the workload of publish/subscribe applications has increased dramatically in recent years, with a million of event messages per second not unheard of [2]. Previous implementations and optimizations based on multi-threaded CPU suffer from various weakness inherited from the Von Neumann architecture. FPGAs has been served as supercomputing platforms for various applications before [3]. The rich on-chip memory bandwidth and stream processing ability in today's reconfigurable logic devices make them a good candidate for accelerating this process. This work explores the architecture and implementation schemes of using FPGA as a pub/sub processor. Major contributions include:

- An architecture for pub/sub matching on FPGA using multiple levels of indices to reduce processing time. The memory requirement and performance of the architecture are analyzed and modeled.
- A scalable multi-core implementation optimized for Xilinx FPGAs. Experimental results show significant improvements in both throughput and power efficiency.

## II. THE PUBLISH/SUBSCRIBE MATCHING ALGORITHM

The followings provides the basic formulation of the pub/sub matching problem. The publishers create **events (e)** to represent the changes in their contents. An event is a list of *Attribute (a)* and *Value (v)* pairs. The subscribers create **predicates (p)** as rules to filter the events for notification. One **subscription (s)** is a set of predicates and one predicates is defined by the *Attribute*, *Operator (o)* and *Value* tuple as described below.

$$
\begin{aligned}
e &= \{< a_i, v_i >: i = [0, el - 1]\} \\
p &= < a, o, v > \\
s &= \{p_i : i = [0, sl - 1]\} \\
p \circ e &\iff a = a_i \wedge o(v, v_i) = true : \exists < a_i, v_i > \in e \\
s \bullet e &\iff p_i \circ e : \forall p_i \in s
\end{aligned}
$$

where $el$ is the number of $< a, v >$ pairs in an event and $sl$ is the number of predicates in a subscription. A predicate $p$ is *matched* by an event $e$, $p \circ e$, if and only if the relation between the values defined by the operator holds true for the attribute. A subscription is *satisfied* by an event, $s \bullet e$, if and only if all its predicates are matched by that event. The input to the system is a stream of events and the satisfied subscriptions per event are generated as the output.

A naive implementation to find all the satisfied subscriptions for a new event is to loop through all subscriptions and for each $p$ inside every $s$, try to find one pair in $e$ that is matched. For a system with $m$ subscriptions, the average number of required comparisons is $(m \times sl \times el)/2$, assuming half of the pairs in $e$ are compared on average. The required computation power and memory bandwidth grow linearly with the number of subscriptions. This is not efficient for two reasons: First, every subscription in the system has to be checked for each event, even if its predicates do not refer to this event's attributes. Second, the predicates included in various subscriptions may be equal to or covered by each other. Thus, predicate operators have to be repeatedly calculated per subscription even if the result of this calculation is already known.

Database systems usually create indices to reduce search time in large data set. The same idea can be applied in the pub/sub matching problem as demonstrated by Fabret et al [4]. The original filtering algorithm in [4] is optimized for CPU processing and cache utilization. We augment the algorithm as shown below to target our reconfigurable platform.

```
bv: one-to-one mapping bit vector for all possible p
for ∀aᵢ ∈ e do
    for ∀p : a = aᵢ do
        if p ∘ e then
            bv(p) ⇐ 1;
        end if
    end for
```

```
    end for{step 1}
    for ∀p : bv(p) = 1 do
        for ∀s : p₀ = p do
            if bv(pᵢ) = 1 : ∀pᵢ ∈ s then
                output s satisfied;
            end if
        end for
    end for{step 2}
```

The algorithm can be divided into two steps by the two outer **for** loops. The first step ensures that only the predicates with attributes that exist in the event are checked and that any predicate is checked at most once. After the first step, a bit in the vector $bv$ is set when the corresponding distinct predicate is matched by the event. The second step ensures that only the subscriptions with matched predicates are checked and that any subscription is checked at most once. Two lookup tables are used in the implementation. Table $T_{a \to p}$ indexes all related predicates by a given attribute. Table $T_{p \to s}$ indexes all subscriptions start with a given predicate.

Even when the number of predicate checked is largely reduced by the above algorithm, the performance is still insufficient for today's requirements where millions of events are generated per second in a systems with millions of subscriptions. Farroukh et al proposed speedup schemes based on parallel threading [5]. By partitioning and distributing the events, predicates and associated subscriptions to threads, the design can achieve over 1600 events per second or around 1.5ms matching time per event on a 2.3GHz Xeon processor. The scalability is limited by the large software synchronization overhead and the available number of active threads.

## III. ARCHITECTURE

After analysing the issues on the CPU based pub/sub matching designs, we believe that realizing and optimizing the system on a reconfigurable platform such as FPGA devices will significantly improve the overall performance as well as being more efficient in energy utilization. Figure 1 is an overview of the pub/sub matching architecture. A hardware accelerated pub/sub system can benefit from the increase in number of parallel processing cores, the low hardware synchronization overhead, the higher on chip memory bandwidth and the efficient bit level manipulation.

In this system, there are $l$ distinct possible attributes, $m$ subscriptions and $n$ distinct predicates. The *match* processing block compares the outputs from the predicate and event data blocks. The *check* processing block will examine the bit vector positions indexed by the output of the subscription data block. Hardware parallelism can be achieved by multiple instances of the two processing blocks.

Figure 1 does not show any improvement over the original software design. To maximize the performance on FPGA, several platform specific issues need to be addressed. These include the utilization of on-chip/off-chip data storages, the pipeline and synchronization between the two steps and the resource allocation between processing blocks. Based on the system and platform parameters, we also create a performance
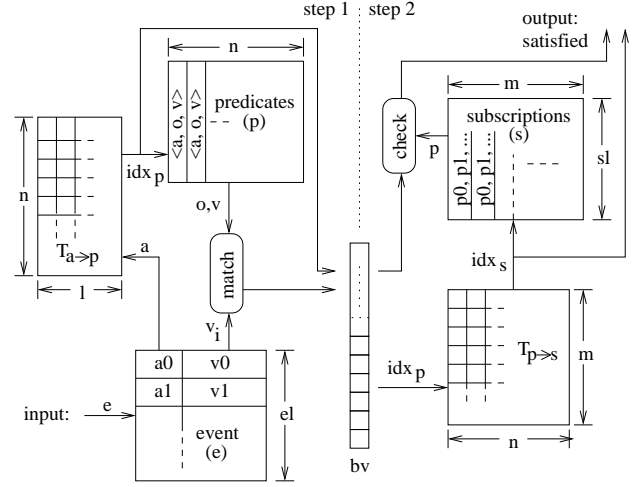


Fig. 1. Pub/sub Matching Architecture.

model which will will help to explore better hardware configuration and greater scalability.

Providing high bandwidth data access, the on-chip memory in FPGA device is a precious resource and must be used wisely. We assume all values are in the range of $[0, v_{max}]$ for both events and predicates. The size of an single event is $el \times (log(l) + log(v_{max}))$ in bits. Since the predicates are indexed by attributes, it is not necessary to store the attribute again inside the predicates. Thus $n \times (log(o_{max}) + log(v_{max}))$ bits are required to store all the predicates, where $o_{max}$ is the number of different operators. Assuming that the subscriptions have an average number of predicates, $sl$, the space for storing all subscriptions is $m \times sl \times log(n)$ bits. Most entries will be empty if a fixed length data structure is used for the two lookup tables. If all records in the tables are stored in variable length, $n \times log(n)$ bits are used for $T_{a \to p}$ and $m \times log(m)$ bits are used for $T_{p \to s}$. An extra level of indices is used to enable efficient access to variable length data structure in hardware. In this level, the starting address and the number of entries in the corresponding record are stored for each possible input. Thus the two tables are $l \times 2 \times log(n)$ and $n \times 2 \times log(m)$ bits in size.

Since the bit vector is the only commonly accessed data structure between the two steps, and the $bv$ must be fully evaluated before checking the subscriptions for each event, the two steps can be decoupled in hardware by using a double buffer structure for $bv$. The predicate matching module continuously checks new events and update $bv$ in one buffer while the subscription checking module continuously examines the bit vector in another buffer. This results in a coarse-grained data pipeline by overlapping the processing time of the two steps. Inside each module, the data path can be further pipelined in a finer granularity to achieved better throughput.

One problem of the multi-threaded software design is synchronization, when each thread updates the global $bv$ from its local $bv$ sequentially. In hardware design, the synchronization overhead is minimized while the scalability is limited by the accessibility of the data. Instantiating $x$ number of parallel matching cores implies $x$ predicates must be read in parallel.

| parameter | symbol | value |
|---|---|---|
| average attribute per event | $el$ | 50 |
| maximum value | $v_{max}$ | 15 |
| number of subscriptions | $m$ | 6,000,000 |
| average predicates per subscription | $sl$ | 10 |
| number of distinct attributes | $l$ | 100 |
| number of distinct predicate | $n$ | 1500 |
| possible operators | $o$ | $=$ |

We then speedup the matching process $x$ times at the cost of wider read port or even multiple copies of predicates when ports are fixed. The same method is applicable to the checking module where $y$ parallel checking cores provide $y$ times speedup and require $y$ times memory bandwidth.

Optimizing the system throughput (event per second) is the primary design goal. Assuming all memory access latencies are hidden by the data path pipeline, we model the processing time for a single event as $T_E$.

$$T_E = max(T_M, T_C)$$
$$T_M = (el \times (n/l) \times T_m)/x$$
$$T_C = (n - \hat{n}) \times T_{bv} + (\hat{n} \times (m/n) \times T_c)/y$$

In $T_M$, $n/l$ is the average number of predicates with the same attribute in a uniform random distribution and $T_m$ is the time for matching $x$ predicates and updating the $x$ bits of $bv$ in parallel. In $T_C$, $\hat{n}$ is the number of '1's in $bv$, $T_{bv}$ is the time for checking a single bit in $bv$, $m/n$ is the average number of subscriptions starts with a same predicate in a uniform random distribution and $T_c$ is the time for checking $y$ subscriptions in parallel. The value of $\hat{n}$ depends on the operators in predicates. Assuming only the equality, $=$, is used, one candidate predicate has $1/(v_{max} + 1)$ chance of being matched. Thus the $\hat{n}$ for equality is $el \times (n/l) \times (1/(v_{max}+1))$.

The actual performance depends on the data set where the distributions may not be uniform. Even for the same data, the order of predicates in subscriptions also has significant impact. Since these issues affect the software implementation as well and may be addressed irrespective of the hardware accelerator, we will not consider them here. The hardware platform itself also affects the accuracy of the model due to non-deterministic external memory latency. This model provides a theoretical upper bound of the system performance regardless of the data set and actual implementation.

## IV. IMPLEMENTATION

We adapt the typical system parameters from [5] in this work as shown in Table I. The target hardware platform is an FPGA device with internal dual-port memory blocks and multiple banks of external memory such as the Xilinx Virtex or Spartan devices. The system is modularized by separating the *match*, *vector* and *check* process with parameterized interface. Thus user can scale the system and adapt to a new platform easily by assembling the modules.

Figure 2 shows the implementation details of the *match* module. The event buffer is implemented in distributed flipflop
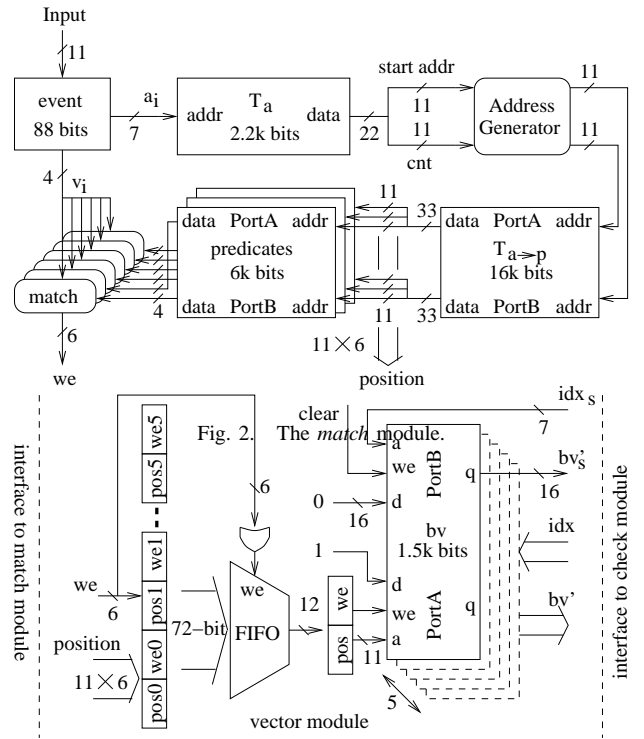


Fig. 3. The *vector* module (single buffer).

as shift register. All other data are stored in 36Kbit Dual Port Block RAM (RAMB36). The $T_a$ table stores the starting address and the number of predicates associated with a given attribute. Based on this information, the address generator generates two address streams for the $T_{a \rightarrow p}$ table which is configured as a dual 36-bit ports memory. The design can provide 6 different predicate indices concurrently by grouping the $T_a$ outputs. After the last address is generated, the circuit starts processing the next attribute in the event register. Unlike the $T_{a \rightarrow p}$ table, the predicates are not sorted by attributes. Thus we need 3 instances of RAMB18 to consume the generated indices and 6 matching operators to match the 6 predicates in parallel. For $x$ parallel output, $1 + \lceil x/6 \rceil + \lceil x/2 \rceil$ Block RAMs are used. Since the module process events in a streaming manner and there is no feedback path or data dependency between attributes, an event can be processed in average 20 clock cycles ($T_m = 1$).

Since the *match* module produces the positions as outputs, it is unwise to implement the bit vector in distributed flipflop as that may require 6 parallel shifters each in $1.5kb$ width. We store the $bv$ in RAMB36 units with the help of an input FIFO as shown in Figure 3. The $< pos, we >$ pairs only enter the asymmetric FIFO if any of the 6 predicates is matched. This scheme eliminates the need of shifters at the expense of serializing the update to $bv$. Based on the fact that only a few of bits will be set when compared to the number of predicates checked for each event (e.g. total 7.5 bits on average in this example), this will not become a bottleneck in the system. The $bv$ vector is stored in a dual port RAMB36 unit where the *match* module and the *check* module use independent ports. *PortB* is also used to reset the vector to zero before a new event
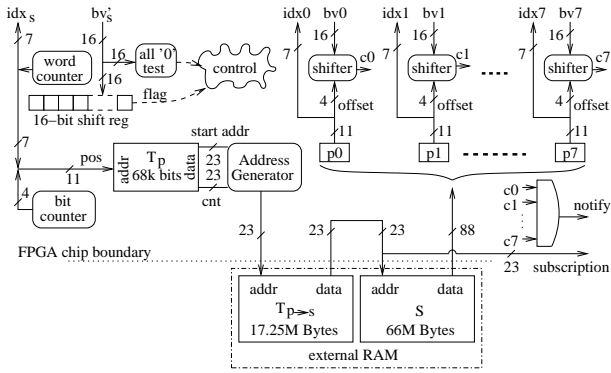
Fig. 4. The *check* module.

is processed. With a 16-bit input, the vector can be reset in 94 clock cycles. To use two buffers as described in Section III, the circuit in Figure 3 is duplicated twice.

In the second step of the algorithm, it requires scanning through the bit vector to locate bits set to '1'. It costs 1500 clock cycles to scan $bv$ in hardware bit by bit. In this implementation, a 16-bit word, $bv'_s$, is checked in one clock cycle. Words with all bits are '0' will be discarded while a 16-step serialized scan is performed only for those non-zero words. When most bits in $bv$ are '0's, the scanning process is improved by a factor of 16. This word-based scheme does not slow down the process even when most bits are '1's. Figure 4 shows the *check* module in hardware implementation.

While table $T_p$ is still implemented in RAMB36 units as in the *match* module, table $T_{p \to s}$ and the subscriptions list $S$ are too large that they must be stored in external memory. We assume that DDR2 memory chips with 32-bit data bus are used. The data bus to user logic will then be 128-bit in width for single edge half frequency clock. Parallelism is achieved by fetching 8 predicate references from the $S$ memory bank in one read operation. The upper 7 bits of a reference are used as the index input to the *vector* module to fetch a 16-bit word, $bv$. The lower 4 bits control the shifter which shifts out the bit being checked. This checking core is replicated 8 times for each predicate reference from the external memory. If outputs from all cores are set to '1', then the all predicates in this subscription are matched by the current event and the subscriber should be notified.

There are 8 extra copies of $bv$ in the *vector* module supporting these 8 parallel checking cores to avoid bus contention. To increase the degree of parallelism in the *check* module, one must first increase the output width of table $T_{p \to s}$ such that multiple copies of subscription list can be indexed concurrently. Then the number of checking cores and the number of extra $bv$ copies in the *vector* module should then be increased accordingly. The interface to the external memory is the largest limiting factor when scaling up the *check* module.

## V. RESULTS

The design is captured in VHDL description and synthesized using the Xilinx ISE 11.4 tool chain. The Xilinx Virtex-6 LX240T, which is available on the ML605 board, is targeted to evaluate the cost and performance. Table II shows the FPGA

TABLE II
IMPLEMENTATION RESULTS.

|  | Virtex-6 | Xeon |
|---|---|---|
| LUTs | 406(0.3%) | N/A |
| FFs | 385(0.1%) | N/A |
| RAMB36 | 19(4.6%) | N/A |
| Freq. (MHz) | 254 | 2300 |
| Avg. Throughput ($e/s$) | 9.5k | 1.6k |
| Agv. Power (W) | 2.031 | 40 |
| Cost (USD) | 3,000 | 530 |

implementation results and compares them with the software performance in [4].

The design has 6 matching cores and 8 checking cores as shown in Figure 2 and 4. The average throughput are computed using the performance model presented in Section III under uniform random inputs. The average power is estimated by the Xilinx XPower Estimator under 75% switching rate. In this configuration, the *check* module dominates the run time. With more external memory bandwidth and more checking cores, the FPGA performance can be further improved. It is difficult to measure the power of an isolated CPU, thus the CPU power is estimated to be half of the TDP value from the CPU specification. In reality, a PC system consumes even more power than a CPU accelerator card.

Even a single engine, as implemented in this work, is 5.9 times faster than the highly optimized software design. It is possible to fit 20 engines in a single FPGA which will be 100 time faster than the CPU system with a small cost of increased memory chips. The FPGA is also more cost effective in terms of event per second per dollar. The available on-chip BlockRAM and the memory I/O will be the limiting the scalability of the architecture. The problem can be solved by using multi-FPGA clusters such as the one described in [6]. With 16 Virtex-5 FPGAs, the cluster can easily handle over 2.5 million events per second and perform more sophisticate match besides equality.

## VI. CONCLUSION

In this work, we develop an architecture for the pub/sub matching engine in FPGA which outperforms software running on high end CPU in terms of throughput, system cost and energy efficiency.

## REFERENCES

[1] P. T. Eugster and et al, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.
[2] Aite Group, "Market data infrastructure challenges," April 2009, www.aitegroup.com/reports/200904222.php.
[3] R. Baxter et al., "Maxwell - a 64 FPGA supercomputer," in *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, 2007, pp. 287–294.
[4] Françoise Fabret et al., "Filtering algorithms and implementation for very fast publish/subscribe systems," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, 2001, pp. 115–126.
[5] A. Farroukh and et al., "Parallel event processing for content-based publish/subscribe systems," in *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, 2009, pp. 1–4.
[6] K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with fpgas and gpus," in *FPGA '10: Proceedings of Field Programmable Gate Arrays*. IEEE Computer Society, 2010, p. to be appear.