Making Search Engines Faster by Lowering the Cost of Querying Business Rules Through FPGAs

Fabio Maschi, Muhsen Owaida
Gustavo Alonso
first-name.last-name@inf.ethz.ch
Systems Group, Dep. of Computer Science
ETH Zurich, Switzerland

Matteo Casalino
Anthony Hock-Koon
first-name.last-name@amadeus.com
Amadeus
Sophia Antipolis, France

ABSTRACT

Business Rule Management Systems (BRMSs) are widely used in industry for a variety of tasks. Their main advantage is to codify in a succinct and queryable manner vast amounts of constantly evolving logic. In BRMSs, rules are typically captured as facts (tuples) over a collection of criteria, and checking them involves querying the collection of rules to find the best match. In this paper, we focus on a real-world use case from the airline industry: determining the minimum connection time (MCT) between flights. The MCT module is part of the flight search engine, and captures the ever changing constraints at each airport that determine the time to allocate between an arriving and a departing flight for a connection to be feasible. We explore how to use hardware acceleration to (i) improve the performance of the MCT module (lower latency, higher throughput); and (ii) reduce the amount of computing resources needed. A key aspect of the solution is the transformation of a collection of rules into a Non-deterministic Finite state Automaton efficiently implemented on FPGA. Experiments performed on-premises and in the cloud show several orders of magnitude improvement over the existing solution, and the potential to reduce by 40% the number of machines needed for the flight search engine.

ACM Reference Format:
Fabio Maschi, Muhsen Owaida, Gustavo Alonso, Matteo Casalino, and Anthony Hock-Koon. 2020. Making Search Engines Faster by Lowering the Cost of Querying Business Rules Through FPGAs. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3386133

SIGMOD '20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3386133

1 INTRODUCTION

Business Rule Management Systems (BRMSs) are used in enterprises to capture changing sets of policies and constraints that affect business logic [4, 13, 17, 20]. Examples of applications include, e.g., checking the eligibility of a customer to particular offers, or identifying fraudulent operations by checking them against known patterns. Very often, BRMSs are embedded as part of larger applications, such as search engines. In such settings, the performance of the BRMS can be a bottleneck for the overall system. In large deployments, where the number of rules involved might be in the order of hundreds of thousands, BRMS can also have a large footprint in terms of the computing infrastructure needed.

In this paper, we focus on a concrete use case from the airline industry: the flight search engine used by Amadeus to provide search services to the travel industry. Like many other search engines, Amadeus' flight search engine is a large scale distributed system comprising many different components. The Minimum Connection Time (MCT) module is one such component and it is implemented atop a BRMS. It is used in the early stages of the search (in the so called Domain Explorer component) to determine the validity of a connection between two flights in terms of the time needed between the arrival of a plane and the departure of the connecting flight. The module plays a key role in terms of the performance and total cost of operating the search engine. When a query looking for flights between a departure and a destination airport needs to be processed, a large number of potential routes has to be computed. For all routes that are non-direct flights, the MCT module is invoked to ascertain the minimum connection time to the next flight. Thus, the MCT module needs to fulfil stringent performance requirements on both latency per query and overall query throughput. Because it is used as a module within a larger system, there are additional constraints in terms of the amount of memory used that determine what type of BRMS can be employed in practice. In the current deployment, the MCT module is responsible for 40% of the computing resources used by the Domain Explorer.

In what follows, we describe our efforts to redesign the MCT module to achieve several goals. First, to reduce the latency involved in computing the MCT and, if possible, to also increase the throughput without having to use additional computing resources. Second, to provide a scalability path at a time where the number and complexity of the rules used in MCT is increasing; while the latency margin for query processing remains constant, since the flight search is an online interactive system. Third, to determine whether the number of computer nodes can be reduced so as to have a more efficient flight search engine. On top of these goals, our solution must respect the very restrictive constraints on downtime for rule set updates imposed by the system.

The solution we propose is based on hardware acceleration through FPGA, enabling both orders of magnitude better performance, as well as a different architecture, where an FPGA used as a co-processor (PCIe attached) can replace several CPU-only computing nodes. The contributions of the paper are as follows:

(i) We explore the problem of embedding a BRMS into a search engine and discuss the many constraints affecting the overall system architecture and leading to using one system over another. We also briefly comment on current implementations, and why it is important to improve their efficiency.
(ii) We describe how to efficiently query large rule collections by capturing the rules using a Non-deterministic Finite State Automaton (NFA). We provide two versions of the design, one for multi-core CPU, and one for FPGAs. We also describe in detail how to exploit the potential for parallelism on FPGAs to achieve high-frequency clock designs.
(iii) We evaluate the resulting system against two baselines: Drools [4], a widely used open-source BRMS, and our own CPU implementation using the NFA structure. We also evaluate the design both on-premise and in a cloud deployment over Amazon F1 instances.
(iv) Our results demonstrate a potential gain for the FPGA version of our design that is four orders of magnitude faster than the current latency threshold, and three orders of magnitude faster than Drools and the CPU version of the same design implemented on the FPGA. In terms of cloud efficiency (queries per U.S. Dollar), our FPGA solution is an order of magnitude better than the CPU version. The results also suggest a potential reduction of 40% in the number of machines needed for the Domain Explorer, as a consequence of the performance gains obtained from the FPGA design. All combined, the high energy efficiency of our solution indicates a reasonable move in the context of environment-conscious deployments.

2 BACKGROUND AND RELATED WORK

In this section, we cover necessary background information on BRMSs, the Amadeus flight search engine, the Minimum Connection Time module, as well as about FPGAs. We also differentiate the results in this paper from those in the literature by pointing out the differences between more general approaches and our use case.

2.1 Business Rule Management Systems

Business Rule Management Systems are widely used both as stand alone engines as well as components of larger systems. Examples of BRMSs include Oracle Business Rules [20], IBM's Operational Decision Manager [13], Microsoft's BizTalk Business Rule Engine [17], or open-source systems such as Drools [4]. These engines capture complex and evolving business logic as a set of rules expressed in a high level syntax. Rule Engines are then used similarly to a database by posing queries that return a decision or trigger an action according to the parameters set in the query.

In a rule engine, the simplest form of matching consists of retrieving the rules in the database fitting the query by iterating over the rules, a process that can be accelerated using indexes [16]. More advanced forms of query processing involve forward chaining and/or backward chaining, where each of the queries is treated as a fact and, in an iterative process, the engine derives new facts until a rule is found; determining the decision or action to be followed. In these cases, more general solutions such as the RETE algorithm [8, 23] are used. Rule engines differ as well in the way they operate. They range from the static, stateless engine that processes queries and delivers matching rules to complex event processing systems that keep state and are constantly fed with events, triggering an action when some particular state is reached [14, 26].

In this paper, we focus on stateless rule matching rather than on the more general inference case. The use case is similar to that of XML processing systems [2, 3, 11]. Unlike the former ones, however, the MCT use case involves a wider variety of data types and matching conditions, as well as having no ordering restrictions on the processing of the attributes. Limiting as these constraints might seem, they are actually met in many situations. Within the airline industry, stateless rules over a collection of attributes are used to determine, for instance, whether a passenger is eligible for an upgrade, whether a ticket can be changed, or whether overbooking is allowed in a flight and by how much. Amadeus alone has a total of 783 use cases of stateless business rule engines used in production. More generally, our approach matches almost the entire eXtensible Access Control Markup Language [22], an OASIS standard for stateless attribute-based access control, used to define and process access control rules of the

Table 1: Example of the rules $r_{[0,5]}$ determining the Minimum Connection Time (actual rules have 22 criteria) and a possible query ρ_0.

	Airport	Time frame	Region	Terminal	Decision	Precision
r_0	ZRH	*	International	*	90 min	Low
r_1	ZRH	*	Schengen	T1	25 min	Middle
r_2	ZRH	Summer '20	Schengen	T1	40 min	High
r_3	ZRH	Winter	Schengen	T1	25 min	High
r_4	CDG	Winter	Schengen	T1	25 min	High
r_5	CDG	Sundays	International	T2	45 min	High
ρ_0	ZRH	12th Aug '20	Schengen	T1		

type "*If the user meets conditions A, B, C, D; and the access meets conditions E, F; then grant access*".

For the rest of the paper, we will use the following notation when needed (see Table 1 for an example). A rule-type t, similar to a database schema, is a structure $t = \langle C, R, D\rangle$, where C is a set of criteria (e.g., *Airport*, *Time frame*, *Region*, and *Terminal*); R, a set of rules; and D, a set of decisions (e.g., *25*, *40*, *45*, and *90 minutes*). A criterion $c \in C$ is a structure $c = \langle \mathbb{A}, \mu, \varpi_c\rangle$ where $\mathbb{A}$, is a set of values — or alphabet (e.g., *Schengen* and *International* for criterion *Region*); $\mu : \mathbb{A}^2 \rightarrow 0, 1$ a matching function (e.g., a numerical range check between the *Time frame* and the query value); and ϖ_c, a precision weight (i.e., how generic or specific the criterion is). A rule $r \in R$ is a structure $r = \langle \chi, d, \varpi_r\rangle$ where $\chi : C \rightarrow_\perp \mathbb{A}$, is a partial assignment function from criteria to values (i.e, a unique combination of values); $d \in D$; and $\varpi_r = \sum \varpi_c$, the precision weight of the rule. A query ρ is a structure $\rho = \langle \chi\rangle$.

Let us consider rule-type $t = \langle C, R, D\rangle$ and a query ρ. We say that a query ρ matches a rule r, denoted $\rho \sim r$, when the following condition holds: $\forall c \in C : \{\mu_c(\chi_c^\rho, \chi_c^r)) = 1\}$. In most related work, the matching function μ uses simple binary (e.g., and, or, xor) or integer (e.g., equal, greater than) comparisons between values. In our case, the matching function is far more complicated as we discuss below.

For queries matching multiple rules — like ρ_0, which matches both r_1 and r_2 rules — a second step is required to select the most precise one, determined by the precision weight ϖ. R' is the set of all rules $r \in R$ that match ρ: $R' = r \in R \,|\, \rho \sim r$. R'' is the subset of R' rules that have maximum precision weight: $R'' = r \in R'' \,|\, \forall x \in R', \varpi_x \leq \varpi_r$. In the context of the MCT module, an offline correctness check ensures that R'' is either empty or contains only one rule, otherwise it would necessarily indicate two contradictory rules. In situations where R'' may be a set of more than one rule, additional precision distinction methods must be proposed, or the system must relinquish the single decision result requirement.

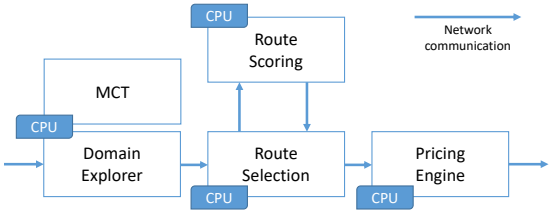


Figure 1: The Flight Search Engine from Amadeus.

2.2 A Flight Search Engine

The Flight Availability Search and Pricing Engine (Figure 1) is a search engine that, given a query specifying an origin and a destination airport, as well as the corresponding dates, lists potential routes between the departure and arrival points with the corresponding flight information and prices. The engine works as an online interactive service and has strict Service Level Agreements (SLAs) in terms of response time and throughput, as it is used by many companies to provide travel services to end customers. Each component has a corresponding latency bound that must be met, so that the overall query processing remains under four seconds. Queries can trigger exploration of a very large space of options, for instance, when a user is flexible on the departure and/or arrival dates.

The engine is divided into several components. The *Domain Explorer* searches the flight domain, exploring all possible connecting airports, carriers and flights combinations; and pre-selects a number of potential routes. These routes are fed into the *Route Selection* component, which uses heuristics and a decision tree ensemble [21] (module *Route Scoring*) to reduce further the set of potential routes most likely to be bought. The reduced set is passed on to the *Pricing Engine*, which then computes the price for each flight combination. The engine is currently implemented in a data centre architecture, where the different components run on different machines, and many machines are used to run the components in parallel to reach the necessary throughput. The system is large, with each one of the components using several hundred machines. In recent work, we have explored the complex interplay between the number of routes being considered and the overall throughput/latency constraints, and we refer to [21] for more details on that aspect of the system and how we successfully used hardware acceleration to optimise the Route Scoring module.

The MCT module is part of the Domain Explorer and runs on the same machines. This design is necessary to cut on the number of network hops needed to process a query, but imposes restrictions on the rule engine. For instance, Drools [4] requires 9.5 GB of main memory to run the hundreds of thousands MCT rules. Such a high resource consumption makes it

impossible to embed it within the Domain Explorer given the existing architecture. Deploying it stand-alone in its own set of servers is not an option either, as it would cause network overhead. As a result, the MCT is a customised implementation in C++ developed at Amadeus intended to provide the required performance while minimising memory consumption. In addition, being part of an online interactive service as the Domain Explorer, MCT downtime for rule set updates is restricted to a minimum. Nonetheless, the current MCT module deployment consumes 40% of the computing resources allocated to the Domain Explorer, giving a good idea of why it is important to make it more efficient.

2.3 MCT: Filtering Impossible Connections

The rules determining the minimum connection time are provided by each airline. The connection time is affected by a number of variables (e.g., airports, terminals of arrival and departure, whether passport or immigration controls are involved, aeroplane model, time of day, etc.). The rules defining the MCT change regularly, so airlines can adapt their flight offer to their most recent logistic and commercial constraints (e.g., temporal peaks in flow of flights or passengers, changes in connection preferences between airlines, etc.). The flight search engine encompasses all airports worldwide, and every airline contributes a long list of rules for every airport where they operate. Currently, the MCT module operates on over 136K rules, and one of the goals of the work reported in this paper is to be able to scale it to 400K rules without performance losses or requiring additional computing nodes. Table 1 shows a simplified, but syntactically representative example of how the MCT rules look like, note that actual rules have twenty-two criteria.

There are several differences in rule management between the MCT module and general-purpose BRMS. Unlike regular expression matching, commonly used in complex event detection over streams, e.g., [14, 27], there are no ordering constraints between the different criteria either for rules or for queries. In some sense, rules and queries, as used in MCT, have the same semantics as tables in the relational model in terms of imposing no order among the attributes. The same distinction applies with work done for filtering XML [3, 11], where ordering is derived from how an XML path is enumerated. Moreover, unlike in stream processing, XML filtering, or complex event detection, the matching criteria used in MCT are very rich and highly data type dependent. Every criterion uses a different alphabet and data type, requiring specialised comparison operators for each one of them. The MCT use case requires support for: ranges (e.g., over dates or flight numbers); wildcards, '*', used in non-mandatory criteria to match any query value; and a variety of specialised data types (terminals, booleans, airline codes, etc.). MCT also uses hierarchical data where elements are used to subsume a long list of other elements [16] (e.g., *Europe* refers to all airports in Europe). The more complex data types are important to simplify the writing of the rules. This can be seen in Table 1, where rules with *Region=Schengen* stand for what otherwise would be the Cartesian product of all the airports within the Schengen area.

2.4 FPGAs

A Field-Programmable Gate Array (FPGA) has traditionally been a matrix of configurable logic that could be programmed using a Hardware Description Language to implement arbitrary logic circuits [25]. Nowadays, an FPGA contains much more than re-configurable logic and is also increasingly being designed to be extensible [19].

An FPGA is a spatial architecture, where designs exploit parallelism through pipelining and redundant instances of the processing elements. This property is what allows FPGAs to operate at line rates, for instance, when connected to the network in a smart-NIC configuration as it is done by Microsoft in the Catapult deployment [7]. In the context of this paper, we will exploit this property to implement an NFA to model the MCT rule engine. The advantage of an NFA on an FPGA is that multiple transitions can be evaluated within one clock cycle. As a result, and even with the lower clock rates of FPGAs (150-500 MHz) compared to CPUs, FPGAs can provide an unprecedented degree of parallelism over other computing platforms. We also increase parallelism through pipelining: we divide the design in stages arranged as a linear data flow, such that stages can process the next query as soon as they finish with the previous one.

2.5 Finite State Automata

Finite State Automata (FA) are used in a wide range of applications: parsing, compilation, text processing, regular expressions, networking, complex event processing, etc. So wide, in fact, that there are proposals to provide support for FAs directly in hardware [5, 6, 18]. From the data processing perspective, there is also a large amount of literature on the differences between Deterministic (DFA) and Non-Deterministic (NFA) FAs. On CPUs, the random memory access patterns caused by the multiple transitions of an NFA affect performance unless the matching operations involved with each transition are expensive enough. This is the case in data processing tasks such as, e.g., parsing and comparing strings in XML processing, where NFAs are the norm as a way to avoid having to deal with a very large number of states [2, 3, 27], since they do not suffer from the state explosion that DFAs are prone to. In our use case, for instance, the number of NFA transitions reaches the order of hundreds of thousands, whereas the same rule set requires several orders

of magnitude more on the DFA version. The amount of effort to generate and process such a large data structure does not pay off given the relative performance improvements it brings (Section 5.2.3). For some different scenarios, however, it has been pointed out that, in some use cases for CPUs, DFAs can be a viable alternative to NFAs for stream processing, as a careful design can avoid the state explosion [11, 15]. We have not been able to apply such techniques to our case to keep the DFA to a manageable size.

Much effort has been devoted to optimise the amount of storage of diverse forms of FAs [1, 11, 14, 24]. A lot of this work has many similarities, differing mostly on how each approach takes advantage of the concrete use case to optimise the FA. In here, we use similar techniques adapted to the MCT use case. For instance, Kolchinsky and Schuster [14] recently suggested techniques to combine multiple patterns into a single NFA for complex event processing over streams. Like many others, they exploit criteria reordering to increase prefix merging, with the resulting NFA taking the form of a tree. By additionally exploiting suffix merging, our NFA preserves the level by level structure of a tree, and is turned into a more efficient acyclic directed graph.

A multitude of research efforts in academia have demonstrated FPGA implementations of various NFA use cases [9, 24, 31, 32]. Sidler et al. [24] propose an FPGA engine for string processing using NFAs. Unlike our design, each pattern (rule) is represented with its own NFA, a common approach [10, 32] when there are many queries and a few rules. In [24], as in our design, the NFA resides in memory and, thus, can be dynamically updated at runtime without having to reconfigure the FPGA. This is a common trade-off in FPGA designs, where hard-wiring the NFA states into logic elements boosts performance [18], but imposes reconfiguring the FPGA for every rule set update, which causes long downtime. In addition, mapping the NFA to circuit logic does not scale well for NFAs with many states. Ganegedara et al., [9] have proposed an automated framework to translate regular expressions into NFAs, and map them onto FPGAs. Their approach generates an NFA for each pattern, and tries to group similar NFAs later on using standard techniques. For a large collections of regular expressions, this approach explodes in terms of resource requirements and does not scale. In our case, rules are simpler than regular expressions and we can exploit several aspects of the MCT use case to simplify the automatic generation of the NFA.

3 A COMPACT REPRESENTATION OF BUSINESS RULES

In what follows, we present step by step the NFA we will use for the implementations, we also explain how we exploit the characteristics of the rules to optimise the final graph.

3.1 Data Structure

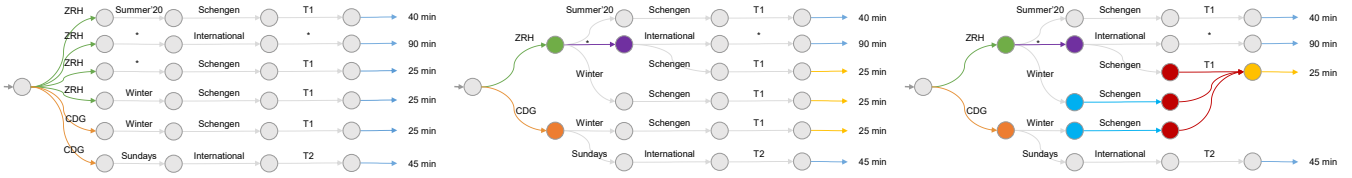
As briefly discussed above, there are many algorithms to construct FAs from regular expressions. In our case, the semantics of the MCT rules and the structure of the NFA we construct make the process much simpler. Unlike most of the work done to date on FPGAs, which typically focuses on processing regular expressions [10, 24, 26, 32], we do not have to deal with NFAs of arbitrary structure. An MCT rule is not a regular expression, and matching is just a conjunction of comparisons. In addition, the criteria can be evaluated in any order, providing more flexibility for optimisations.

Nodes in the NFA correspond to a state in which the query being evaluated has matched the path from the root to that state. Transitions correspond to possible values for the corresponding criterion in each level. For the moment being, we assume the criteria are processed in the same order as they are listed in the rules/queries. We start with a single root — or origin node. From this, the NFA is first constructed as a tree. For each rule, a full path to a leaf is constructed by adding a transition for each criterion labelled with the value indicated in the rule. Between each transition, an intermediate node is added. The value at the leaf is the minimum connection time implied by that rule. Once this is done for all rules, the NFA consists of a root node from which as many paths emanate as there are rules (Figure 2.a). Although the final NFA will not be a tree (Figure 2.c), it retains a level by level structure, where each level in the tree corresponds to one criterion in the rules. The resulting graph has the following properties:

- All paths share the same root;
- Every level correspond to a single rule criterion, hence the depth is equal to $|\mathbb{C}|$;
- Multiple states may be active (valid paths) at the same time;
- There are L leaf states, where $L = |\mathbb{D}|$ is the cardinality of the alphabet of decisions (in our case, the number of possible connection times across all rules);
- It is an acyclic directed graph traversed by navigating from the root to the leaves;
- A matched rule is a continuous valid path between the root and a leaf — and vice-versa.

3.2 Optimisations

Starting from the constructed tree (Figure 2.a), we perform three types of optimisations: forward path sharing (or prefix merging); backward path sharing (or suffix merging); and changing the order in which the criteria are considered. The order in which these optimisations are applied matters. We determine first the order in which the criteria should be evaluated, then perform prefix merging, and finalise the process with suffix merging. These optimisations resemble approaches often used in practice. Reordering the criteria is



(a) Initial tree (one rule = one path)'. **(b) Prefix merging.** **(c) Suffix merging.**

Figure 2: Several steps in the creation of the NFA from a rule set showing (a) the initial tree; (b) the process of forward path sharing; and (c) the process of backward path sharing. The final NFA will merge the three red and two blue states and corresponding paths in the middle level of (c) through suffix merging.

also called push-down of operators when processing XML or streams [12, 14, 27], although, in our case, we have more freedom than the use cases we are aware of, since we can consider criteria in any order. Prefix merging, also known as sub-graph elimination, is also common and often used in combination with reordering (i.e., reordering is done so as to facilitate path sharing). Suffix merging is far less common, since it merges paths to the leaves and, unless some measure is taken to prevent it, makes it impossible to know how a final state was reached. In our case, since what is of interest is the decision value, merging paths is a viable strategy, as it allows to reduce the size of the graph even further.

In the forward path sharing phase (Figure 2.b), same-value transitions leaving the same state are merged, as well as their original destination states. The algorithm navigates the graph (at this point still a tree structure) breadth-first starting from the root until there is no more merging possible in the current level. Once this is achieved, it proceeds to the next level. In the backward path sharing phase (Figure 2.c), states sharing the same path to a leaf are merged. We start at the leaves and proceed backwards until no more merging is possible. With this step, the initial tree adopts a graph structure, since distinct branches may be merged by their leaves or common paths to their leaves.

Reordering the criteria in the MCT case plays a bigger role than in the use cases considered in the literature. When processing regular expressions, the NFA is operating over the same criterion, i.e., all states and the entire NFA works on a single alphabet. In our case, every criterion has its own distinct alphabet. As a result, reordering not only facilitates path sharing, but also has a substantial impact on the space needed depending on the relative cardinality of each criterion, which varies significantly among criteria.

Since we have no constraints in the order in which criteria should be considered, the number of possible permutations is $|C|!$. In MCT, $|C| = 22$ so the potential number of permutations is on the order of 10^{21}. Given that the NFA must be regenerated on a regular basis, we currently use two heuristics to decide on the ordering of the criteria when building the graph. The first one focuses on memory consumption, while the second one focuses on navigation latency. As a baseline for comparing the different approaches, we take an NFA built using a random order for the criteria.

The fixed parts of our NFA are the root and the $|D|$ leaves, they remain constant regardless of criteria ordering. In general, a transition in the first level of the NFA would be shared by all rules with the same value in that criterion. However, transitions with the same value are likely to be duplicated in intermediate levels when the prefix of their paths is not the same. Therefore, for a given number of rules, and assuming uniform distribution of values in each criterion, the probability of having the same value for criterion c_1 will be higher than for criterion c_2 if $|\mathbb{A}_{c_1}| < |\mathbb{A}_{c_2}|$, in other words, if the cardinality of the alphabet in c_1 is smaller than that in c_2, then it is more likely that c_1 transitions will be shared by more rules. By placing c_1 *before* c_2, we maximise prefix sharing and reduce duplication in the next level, thereby reducing the size in memory of the NFA. This is similar to pushing down selection and projection in a relational system. The idea is captured by the following heuristic:

Heuristic 1. *(H1_Asc) Let the rule-type* $t = \langle C, R, D \rangle$, $\forall c_a, c_b \in C : a < b \Leftrightarrow |\mathbb{A}_{c_a}| \leq |\mathbb{A}_{c_b}|$, *where* a *and* b *are the level position (depth) of the criterion within the NFA.*

For Heuristic 1, the bigger the differences in alphabet sizes (and the presence of attributes with very large alphabet sizes), the more efficient the heuristic will be, as the amount of redundancy removed will be larger. Given the symmetry of the path-sharing optimisations, this heuristic also applies in the reversed order. Sorting criteria from the biggest to the smallest alphabet causes a similar effect for backward path sharing. Yet, the actual merging depends on the minimum connection time for the rule, so sharing is less effective, as shown in Figure 3, where Ascending order (H1_Asc) merges more states, but has more transitions and a far bigger worst-case navigation latency. Descending order (H1_Desc) has also lower memory utilisation, so we conclude it is a better option than ascending order when suffix merging is possible.

Figure 3: Comparison between optimisation heuristics: number of states; number of transitions; worst-case navigation latency; memory size (in MB); and the amount of UltraRAM required for FPGA designs.

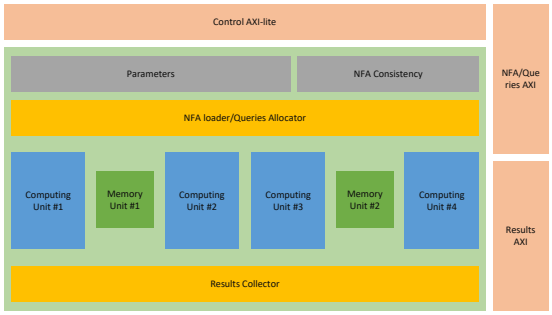


Figure 4: NFA Evaluation Engine (NFA-EE) overview.

Overall size is not the only relevant metric on an FPGA design. Worst-case latency also matters and that is determined by the maximum number of cycles that are need to process a query in the worst case. Thus, in the optimisations, we consider a second metric in the form of the time needed to navigate the graph and reach a leaf. Additionally, wildcard '$*$' values and numeric ranges activate a bigger number of transitions than strictly equality checks. During evaluation, all these transitions will be active at the same time. The larger the number of active paths, the more resources remain busy. Most of these potential matches will be discarded later, so they are wasted work and delay the pipeline. It is possible to reduce the number of potential matches by filtering paths as early as possible. The effect is similar to executing high selectivity operators first in query processing, as that reduces the amount of work needed in later stages. Wildcard operators have low selectivity, while mandatory criteria have high selectivity; so we place those first. Additionally, the first criterion can be indexed, so placing first a mandatory criterion with a big cardinality reduces the evaluation time. This is accomplished with the following heuristic:

Heuristic 2. *(H2_Asc) Let the rule-type $t = \langle C, R, D \rangle$, the order of criteria $c_k \in C$ in the NFA is defined by the following rules: i) mandatory and strictly equality criteria are placed first; ii) criteria evaluating numerical ranges are placed last; and iii) the remaining criteria are placed between the two groups. The order of criteria within each group is defined by Heuristic 1.*

In Figure 3, we show for each one of the approaches we consider: the number of states; number of transitions; worst-case number of transitions to traverse (i.e., sum of the maximum fanout per level); memory size (in MB); and amount of UltraRAM [28] required by a single NFA-EE with two NFA-CUs on FPGA. Note the different scales for each one of the measurements. As shown, the H2_Desc heuristic has the lowest worst-case latency to traverse the NFA, even if it requires more memory (as it has more transitions) than the other approaches excepting the random one. Since, in the context of the MCT design, latency is at a premium, we choose H2_Desc over the other options, as the final size of the NFA is still small enough to fit into the FPGA as needed. On an FPGA, the mapping of memory usage to UltraRAM is not strictly linear, as the NFA is divided into levels; the UltraRAM instances have fixed size; and they cannot be shared among NFA levels. Therefore, memory consumption (be in MB for CPU implementations, be in UltraRAM for FPGA-ones) and worst-case latency must be both taken into consideration, as optimising one does not necessarily mean to optimise the other one as well. For instance, H2_Desc requires 25% more UltraRAM utilisation, but is 32% faster than H1_Desc.

The two heuristics consider only the cardinality and data type of the alphabets. We leave it for future work to exploit the frequency distribution of the values and the paths to optimise the graph even further. The process just described for NFA building from Business Rules needs around 7 minutes on a laptop to create a 136K rule NFA. This is sufficiently fast given the current requirements, even with the planned growth in the number of rules, as the NFA update is performed offline, while the search engine remains operational.

4 BUSINESS RULES ON AN FPGA

In this section, we describe the architecture strategies for an efficient FPGA design of the engine. We further detail how each main component relates to the achieved parallelism degree. We focus on FPGAs configured as accelerators and connected to the host machine through a PCIe interface, such as those provided by Amazon in the AWS F1 instances.

4.1 NFA Evaluation Engine

Figure 4 depicts the architecture of the *NFA Evaluation Engine* (NFA-EE). It has a similar organisation as the Decision Trees Inference Engine proposed by Owaida et al. in [21], as

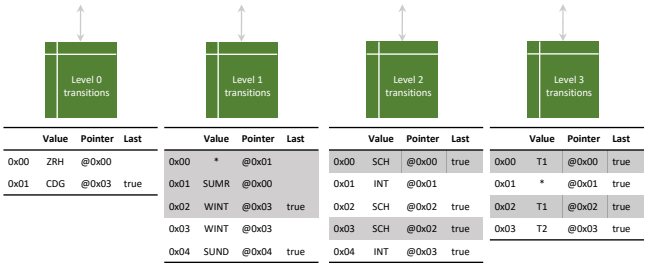


Figure 5: NFA Memory Unit (NFA-MU) overview.

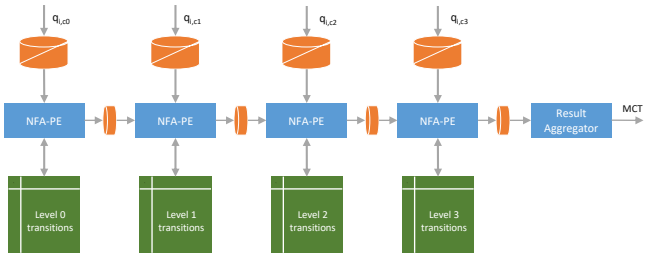


Figure 6: NFA Computing Unit (NFA-CU) overview.

eventually the two engines might be combined (see Section 6). The NFA-EE is highly parallelised, as it contains several *NFA Computing Units* (NFA-CUs), the units implementing the actual NFA navigation logic. Each pair of them accesses one copy of the NFA transitions stored in a shared UltraRAM *NFA Memory Unit* (in Figure 4, NFA-CUs #1 and #2 work from NFA-MU #1, NFA-CUs #3 and #4 work from NFA-MU #2, etc.). Being a scarce resource, memory components are likely to be the limiting factor for scaling the engine up. Allocating one NFA-MU for each pair of NFA-CU allows the engine to double its computing capacity for a fixed memory utilisation. This is possible because FPGA's internal memory is dual-port, so two NFA-CUs can fetch NFA data in parallel and independently from the same NFA-MU.

Processing proceeds as follows: the *Query Allocator* fetches incoming queries and allocates them to available NFA-CUs. Within each NFA-CU, queries are enqueued into the corresponding pipeline. Once processed, results from the query are produced in the same order as the queries arrived. The merging of the results of all the queries ran in parallel is done by the *Results Collector*. The NFA-EE is in one of two main states: *Setup* or *Execution*. In the former, a transitional state, the engine fetches the NFA data generated offline and stores it in the internal memory. This step takes about 500 μs, and is required only once per NFA version. In the case of MCT, the NFA is updated daily, which makes a 500 μs downtime quite affordable and two orders of magnitude faster than the current deployment. During the *Execution* phase, the engine is in its working state and performs query evaluation.

The *NFA Consistency* module automatically detects the NFA version of incoming queries and turns the engine into *Setup* state when necessary. Externally, two AXI-controllers, *NFA/Query* and *Results*, are responsible for loading the NFA data during setup, fetching queries and dispatching results. An AXI-lite controller manages control registers for synchronisation between host CPU and FPGA kernel. AXI-controllers are modules provided by Xilinx Vitis[1] to manage accesses to either the FPGA's DDR or to the streaming interface, as well as to memory registers.

[1]http://www.xilinx.com/products/intellectual-property/axi_emc.html

4.2 Memory Unit: Storing the NFA

To store the NFA in memory, we take advantage of the fact that the NFA is organised by levels, with each one of them corresponding to one criterion. The graph data is organised in such a manner that all the information is contained within the transitions, so they are the single element required to be stored (Figure 5). Within each table, transitions are stored in a breadth-first order, so the ones leaving the same state are contiguous in memory. Each row contains the information required for evaluating a transition as part of a path: (i) the value (or pair of values for ranges) to be matched against the query; (ii) a pointer to the row in the table of the next level where the transition leads to; and (iii) a *last* flag indicating the end of the current set of transitions for that state. For non-mandatory criteria whose function μ is equality, wildcard '$*$' transitions are stored first within their set. This allows an early termination of the evaluation as soon as two matches are found within the set. In the last table, pointers are a reference to the decision value of the matched path.

4.3 Computing Unit: Evaluating queries

The organisation in stages is intended to support a pipelined query evaluation within the NFA-CU (see Figure 6). The pipeline has as many stages as there are criteria in a rule/query. To process a query ρ^i, all of its criteria $\rho^i_{c_0}, \rho^i_{c_1}, ..., \rho^i_{c_n}$ are assigned in parallel to their respective pipeline stages (called *Processing Elements*, NFA-PEs) via FIFO buffers. The execution of a query starts from left to right, with the first NFA-PE processing the first level of the NFA. To do so, it takes the value for that criterion provided by the query and looks for matches in the respective table. When a match is found, a message with the transition pointer is transmitted to the next NFA-PE, which can then start the evaluation of the next stage by looking for matches between the value provided by the query and the set of values listed on the table for that state. The NFA-PE knows when it has looked at all the transitions from one state when it finds the *last* flag.

The nature of the NFA comes here to play. As soon as a match is found at level i, level $i+1$ is informed, and the NFA-PE$_{i+1}$ may start processing the state pointed by the matched

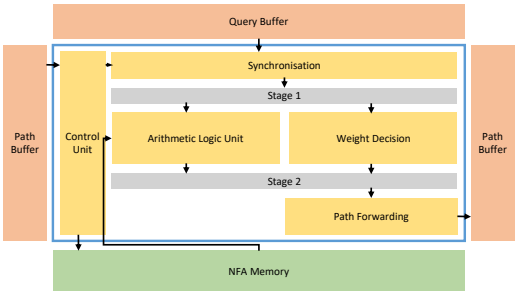


Figure 7: NFA Processing Element (NFA-PE) overview.

transition. Meanwhile, level i continues evaluating other transitions, and emits a message for every match, until it finishes processing all possible transitions at that level for the current query. At this point, level i can start processing the next query by taking it from the queue. Since the number of messages from one level to the next one varies with the query and level, as each different path will possibly lead to a different number of matches to process; FIFO buffers are placed between NFA-PEs to reduce back-pressure, allowing for different progress rates among the NFA-PEs. When the last stage finishes processing a query, it passes the pointer reference to the minimum connection time found to the *Result Aggregator*, which will then select the most precise matched rule for a given query.

With this design, we exploit several forms of parallelism, even if each NFA-PE is sequential: matches at different levels are processed in parallel thanks to the pipeline arrangement; several queries can be in the pipeline at the same time, as when each stage finishes one query, it can start with the next; two computing units are working in parallel processing different sets of queries using the same memory unit; and several pairs of computing units can be deployed within the same FPGA.

4.4 Processing Element: Checking criteria

The evaluation of a specific criterion is done by an *NFA-PE*. Its high level architecture is shown in Figure 7. The *Synchronisation Unit* is responsible for detecting the end of a query evaluation and for fetching the next one. The *Arithmetic Logic Unit* performs the actual matching operation μ_c, while the *Weight Decision* module computes the precision weight of the current path according to the transition under evaluation. The interim precision weight (ϱ) of the path after evaluation in level c is $\varrho_c = \varrho_{c-1} + W$, where $W = \varpi_c$ when the match is against a proper value and $W = 0$ when the match is generated by the wildcard '$*$'. Matched transitions generate a new path message in the *Path Forwarding* unit towards the next *NFA-PE*. The logic that fetches a row from the memory, does the comparison, triggers a message, and moves to the next row is implemented as a state machine managed by the *Control Unit*.

While for software programmers the architecture might seem contrived, it is easy to map it to a modular software design. Each NFA-PE corresponds to the logic needed to understand the data type and comparison operators specific to each criterion (i.e., each NFA-PE is specific to a criterion). Then, the NFA-PEs are connected as a pipeline to form a computing unit, NFA-CUs organised in pairs around an NFA-MU, and pairs of NFA-CUs plus one NFA-MU used as unit of deployment. The use of a table per level decouples each NFA-PE from the other and limits the amount of data that a NFA-PE needs to manage to that level of the NFA. This allows to use the local, fast memory available on the FPGA. It also explains the need to make the NFA as compact as possible, and to minimise the time it takes to navigate the whole pipeline for a query, what we called worst-case navigation latency in Figure 3. The shorter the worst-case, the faster the query is processed. Another way to explain why we used heuristic H2_Desc is that it gives us the shortest time to process a query and the resulting NFA is small enough, so that each NFA-PE can fit its data into local memory.

4.5 Deployment on an FPGA

The NFA-EE design takes advantage of the multiple clocks available in the shells. The data clock, used for AXI-controllers, is usually fixed by the platform. We use a second clock to drive the main kernel, which imposes less constraints and therefore can achieve higher frequencies. The deployment shell used in AWS F1 instances[2] fixes the data clock to 250 MHz. For on-premises implementations, the data clock is fixed to 300 MHz, and the kernel clock is dynamically determined by the framework a value up to 500 MHz, varying according to the complexity of the design (resource utilisation, physical location of UltraRAM instances, etc.).

We use Xilinx Vitis [30] as development and integration platform, which imposes certain design decisions. In particular, the communication interface available on-premises and on AWS boards diverges. The version we used for cloud deployments follows a similar procedure as the one used for GPUs: memory is allocated in the co-processor, the host loads the input data into it, the co-processor processes the data, and the host fetches the output. This procedure is not optimal for FPGAs, as it imposes a high latency overhead for short jobs, and prevents streaming. The new Xilinx shell [29], still in beta version and with restricted access to select customers at the time of writing, provides a new streaming interface, which allows a finer grain synchronisation between the host and the FPGA. We have had access to such an interface and

[2]https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md

experiments on-premises in this paper are reported using this shell. Comparing both interfaces on the same board and environment, we can report that the new one significantly improves latency when the data transfer involves a small amount of data.

5 EXPERIMENTAL EVALUATION

In this section, we present performance comparisons between our FPGA-based Business Rule Engine, and several reference baselines, including a CPU implementation of the NFA. To validate the heuristics used, we also generated and deployed an NFA for each one of the heuristics considered, evaluating their operational performance.

5.1 Experimental Setup

Table 2 in Appendix lists the platforms used for the experiments and their main characteristics. In the cloud, we use both a large, CPU-based server and a smaller, FPGA-based instance. On-premises, we use two large multi-core servers similar to the ones used in the flight search engine, and a modern FPGA board that provides more resources than the one available in the Amazon F1 instance.

The workload used in the experiments is based on the actual workload, augmented with synthetic queries to ensure coverage of the entire rule search space. Experiments are run using batches containing 1 to several thousands queries. Query batching is used in the MCT module given the nature of the flight search engine. When a user looks for a flight, the request that is passed to the MCT module is a batch of possible connections for that user query. Ideally, one would like to check as many routes as possible for a user query to maximise the chances of finding the best ones. How many routes are checked is limited by the time it takes to process a batch, which depends on the batch size, and the overall latency budget. The use of batches is, thus, relevant not only for comparing the performance of our design and of the baselines, but also regarding the integration of the FPGA-based MCT into the search engine (see Section 6). Per batch size, we measured one thousand samples. All the numbers presented correspond to the 90^{th} percentile of the experiments per batch, as that matches the SLAs of the search engine.

5.2 Baselines

We use two baselines as a reference for the performance of our FPGA-based design: (i) the CPU implementation of our NFA structure; and (ii) Drools, a complete BRMS used by Amadeus in other applications. We also discuss the current deployment of the MCT module as an architectural baseline.

5.2.1 Current MCT Implementation. The current MCT module is a C++ implementation that uses neither an FA nor any special processing technique beyond ordering of the criteria and data partitioning. The static rule set is stored in main memory, where it takes about 130 MB. The downtime required to upload an updated set of rules is in the order of hundreds of milliseconds.

The MCT module is embedded in the Domain Explorer of the flight search engine, which maps each user query to a thread that processes this one in its entirety while at the Domain Explorer component. The MCT module is therefore also single threaded. Scalability is accomplished by replicating the MCT module within each processing thread. Since a user query arriving to the Domain Explorer is mapped to a thread, it is easy to see how the requests to the MCT module are batched. Typically, an MCT request will contain a batch of all the MCT queries needed to check all routes being considered for a particular flight search query. Thus, to reduce calling and data transfer overhead, the MCT module is called to perform its task on a batch of queries rather than being called once for every query.

The rules used in the current implementation of the MCT module are partitioned first by airport and then by connection type (combination of *International* and *Domestic* values), two mandatory criteria. This helps to reduce the number of cases that need to be checked for other criteria. The evaluation starts scanning the set by weight group, so more precise rules are on the top of the list. The first rule matching all the criteria of the query is then returned.

5.2.2 Drools. Drools is an open-source Java BRMS supporting both forward and backward chaining-based inference. Drools' pattern matching algorithm, born as an implementation of RETE [8], evolved over the years to increase both performances and rule expressiveness by introducing features like node sharing, alpha/beta indexing, sub-networks, and lazy evaluation [23]. Drools patterns support, on top of conjunction and disjunction of Boolean predicates, existential and universal quantification, negation and computation of aggregates (e.g., count, min, max, average, etc.).

As most RETE implementations, Drools compiles rules' constraints into a tree structure called Alpha Network[3]. Each node in the tree represents a constraint, and any path from the root to a leaf represents a rule. When possible, Drools merges nodes that hold the same constraint while building the Alpha Network, similarly to the aforementioned prefix merging optimisation strategy. A hash index is also built for all the sets of more than three nodes that hold an equality check against a literal on the same fact attribute.

A direct consequence of above node sharing and indexing strategies is that sorting constraints according to Heuristic 2

[3]A second, more complex data structure, called Beta Network, is generated with more expressive rules involving, e.g., multiple patterns, join constraints, universal quantification, negation, aggregates. This is not the case, however, for the kind of rules discussed in this paper.

constitutes a good choice for Drools too: having mandatory criteria first with a consistent order favours node sharing. Moreover, for the same reasons discussed in Section 3.2, sorting criteria by ascending cardinality of their alphabet optimises for smaller number of nodes. Since Drools only provides prefix merging optimisation, descending order does not bring any advantage. Finally, numerical range checks, not being indexable with hash tables, are better placed at the end, favouring the construction of indexes on the equality constraints that come before. We therefore used this heuristic to model MCT rules in Drools.

When compiling the full set of 136K MCT rules, Drools generates an Alpha Network of 566,190 nodes in about 2 minutes, which requires about 3.5 GB to be stored in the Java heap. Before running the actual batch, we perform a warm-up run, where every query is evaluated multiple times. This helps mitigate the effect of lazy static initialisation, which have a non-negligible impact on the first queries, but are not relevant for long-running server applications such as MCT. The number of warm-up queries needed to reach the steady-state response time mainly depends on the size of the network: for MCT, we empirically found it to range from 10,000 times the batch size for small batches to 10 times the batch size for big batches.

For every query, we measure the time required to perform operations: (i) insert in a Drools session a MCT fact with the query input; and (ii) run the engine with a limit of maximum 1 rule execution. Finally, we retract the fact from the session, so that it is ready for the next query. We chose not to account for the time required to retract the fact from the session because this can be done in parallel to sending the MCT result to the client. We ran the benchmark on Oracle HotSpot 1.8.0_65 allocating 16 GB of heap to the JVM. We experimented with smaller heap sizes as well, and found that below 6 GB the garbage collection activity begins to significantly impact performance. With a heap size of 6 GB, the virtual size of the process settles at 9.5 GB and the resident-set size at 7.2 GB.

Being single threaded, we compare the execution time of Drools to our CPU single threaded baseline (Figure 11), however for throughput comparisons we compute what would be the throughput of a server deploying the solution, hence the 80-threaded CPU baseline. For small batches (up to 1,024 queries), Drools is about five times faster than the CPU-NFA design. This, at the cost of using 3,200 times more memory, which is an important limiting factor for deployment in production of the MCT use case.

It is important to understand the reason why Drools needs 9.5 GB of memory: this engine is based on a modern version of RETE, an algorithm that improves performance at the cost of increasing the memory utilisation, a common trade-off to many rule engines. In all fairness, though, the RETE algorithm is more powerful than what is needed for the MCT use

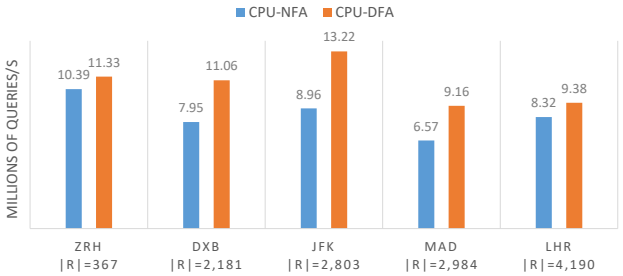


Figure 8: Throughput comparison of NFA and DFA implementations on CPU for five small rule subsets.

case, as it supports inference and complex event processing. Drools is also implemented in Java, which adds to the overall memory requirements and performance overhead. In addition, the MCT workload dimension imposes a long downtime for Drools on every rule set update, which is not acceptable for an interactive service as the fight search engine.

5.2.3 CPU implementations. We have developed two CPU versions in C++ using a DFA and an NFA similar to the one used in the FPGA. Both use Heuristic 2 with descending order to optimise and reduce the size of the FAs. Both index attributes as needed to speed up access. Five small data sets, corresponding to rules from individual airports, were used for the DFA vs. NFA comparison. We have tried to generate DFAs for larger subsets of rules, but we were not able to do so even using a very large server. Our analysis show that, for 136K rules, the intermediate DFA would reach about 3.5 billions of transitions, and an optimistic estimate of the size after suffix merging would be about 1 billion transitions. Figure 8 shows that the CPU-DFA is about 10% to 47% faster than the CPU-NFA, at the cost of having a larger memory footprint. This cost is about 300% to 500% bigger than the NFA-equivalent data structure, and is likely to increase as the number of rules increase.

The CPU-NFA implementation follows the model used in the current system: each MCT module runs in a Domain Explorer thread, and each one keeps its own copy of the NFA. This makes the design compute-bound. Each thread navigates the NFA using a Depth-First-Search strategy. Since the NFA takes less than 2 MB (Figure 3), it largely fits in the cache. Furthermore, the NFA data is read-only during evaluation, eliminating any sort of synchronisation and cache-invalidation among different cores. This design sacrifices parallelism within the MCT module, but allows to scale the Domain Explorer by spawning a thread per user query. The CPU version of the NFA is an important reference, as it is faster than the current MCT implementation, but not as fast as Drools (Figure 11). However, these three systems are algorithmic- and architecturally different among themselves

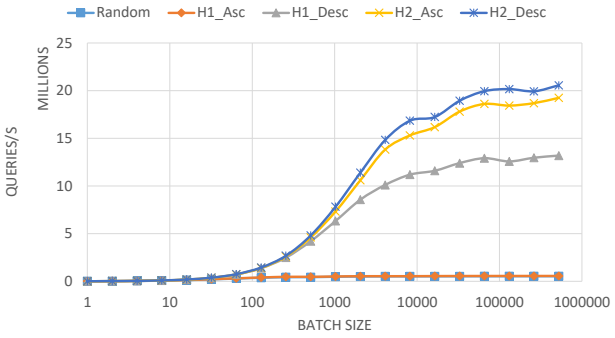


Figure 9: Throughput comparison for the different optimisation heuristics on FPGA using one NFA-EE.

and to what we propose in the paper. The CPU version provides a better perspective of what the CPU and the FPGA contribute, as it implements exactly the same algorithm as that of the FPGA.

5.3 NFA optimisation heuristics evaluation

We now experimentally validate the choice of heuristics used when constructing the NFA (Figure 3, Section 3.2). From the exact same data set, the 136K MCT rules, we generate five different NFAs: two per heuristic (i.e., ascending and descending variations), and one random-order baseline. The results are shown in Figure 9, which measures the throughput reached for each option. The test is done with a single NFA-EE with two NFA-CUs. Clock frequency of all deployments is comparable, ranging from 471 to 500 MHz.

For small batches up to 32 queries, the communication overhead dominates, so disparities due to the different NFAs are not visible. For batches larger than 100 queries, however, H2_Desc achieves up to 20M queries/s using 2.30 MB of memory, while H2_Asc achieves about 90% of this throughput using about 82% of memory. H1_Desc performs about 65% of the processing rate of H2_Desc, while requiring 70% of main memory to store the NFA. Note that the throughput figures in Figure 9 capture the expected behaviour based on the worst-case latency shown in Figure 3. All these three heuristics place the same criterion as the first to check, being the mandatory criterion with the largest cardinality. The evaluation of this criterion is through an index and, thus, requires only one cycle. For the other approaches, the evaluation of this criterion alone already requires from 1 (best case) to 3,486 (worst case) cycles. This is the reason why H1_Asc and Random ordering perform so poorly, with only 3% of the throughput of the fastest variation. Therefore, when comparing the FPGA design to the baselines, H2_Desc will be the optimisation heuristic used.

5.4 FPGA

To evaluate the FPGA design, we have performed a number of experiments both on-premises and in the cloud, comparing the performance and cost of the different designs. As in all previous experiments, measurements of response time and throughout are as seen from the invoking CPU (i.e., what the Domain Explorer will see). The cloud deployment is as follows: on the host/CPU side, a single-thread CPU kernel manages the data, packs the requests into a batch, allocates the memory in the FPGA's DDR and transfers the batch. Next, the FPGA kernel is called with a pointer to the query batch. Once the FPGA kernel has finished processing, the host fetches the results. The on-premise setup takes advantage of the new streaming interface: the CPU kernel packs the requests into a batch and manages two streaming threads, (i) input data; and (ii) results. The FPGA kernel starts processing as data arrives from the stream, and outputs the results as soon as they are ready. Execution time is measured from the point where the host is ready to start the request until it has received all the results (i.e., it has fetched them from the FPGA's DDR to hosts main memory via PCIe, or has received all the results from the stream).

5.4.1 On-premise experiments. The first set of performance comparisons is on-premises using the platforms listed in Appendix Table 2. Figure 11 shows the execution time of a batch and throughput as a function of the batch size. The plots compare the two baselines (Drools and CPU-NFA), and four versions of the FPGA implementation, using one, two, four and eight NFA-EEs; all deploying the new Xilinx QDMA shell. The plots illustrate the effect of the batch size on performance. FPGA communication overhead over PCIe puts a lower bound on the FPGA response time. This favours the two CPU-based baselines, which are quite fast for very small batches. For large batches, however, the FPGA has enough workload to leverage its huge compute capacity, and compensates for the overhead of PCIe communications.

To understand in more detail this behaviour, in Figure 10 we break down the overheads for different parts of invoking an FPGA kernel using the traditional XDMA shell. Up to 32

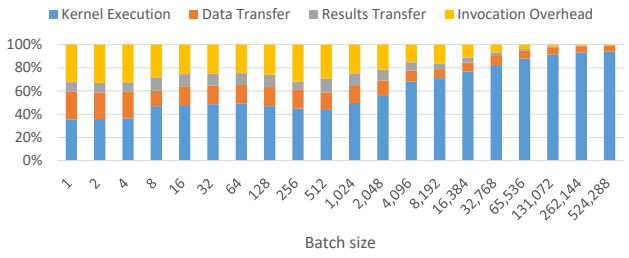
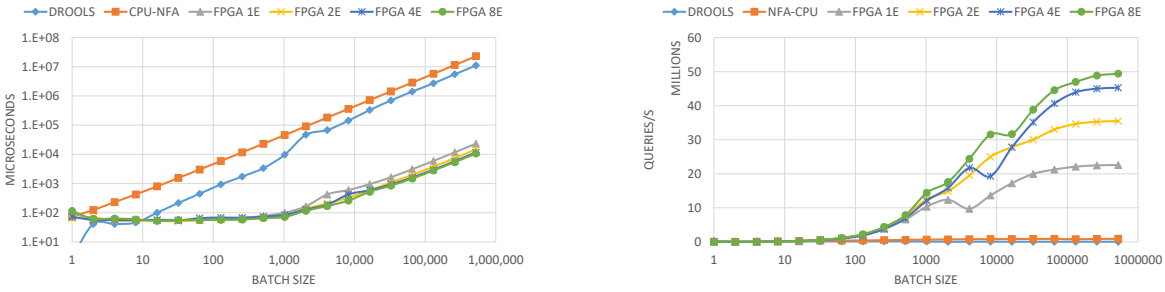


Figure 10: FPGA's response time distribution.



(a) Execution time in microseconds

(b) Throughput in queries per second

Figure 11: Execution time and throughput for the baselines and the FPGA designs with multiple NFA-EEs as a function of batch size. All experiments on-premises.

queries per batch, the invocation and data transfer overheads over PCIe consume more than half of the total execution time. For larger batches, the kernel execution (computation only) consumes close to 90% of total execution time. In the context of MCT, the batches are large enough, so we are not concerned about the PCIe overhead. It must be mentioned, however, that connectivity to accelerators is a very active area of research and development where, in the near future, significant improvements will be available in terms of both bandwidth and latency (e.g., CLX [4]).

5.4.2 Cloud-based experiments. We have conducted a number of experiments in the cloud comparing the efficiency (queries per U.S. Dollar) for each of our designs: the CPU-NFA version and four different deployments of the FPGA implementation, using one, two, four and eight NFA-EEs. We used the AWS F1 instances listed in Appendix Table 2. For our computing cost calculations, we selected two different batch sizes: 128 and 8,192. For the CPU implementation, we count the total machine throughput by multiplying the throughput of a single thread with the number of cores available. This metric is similar to those used by the flight search engine today, and indicates the maximum theoretical throughput the implementation would get if completely utilised for MCT. The experiment shows a query-efficiency per U.S. Dollar of almost one order of magnitude higher for the FPGA solution compared to the CPU version.

Given the different clock frequencies between the FPGAs, shell variations, and the PCIe bandwidth available between host and FPGA, on-premises deployments have a throughput 30% higher, on average, than in the cloud. But such a comparison is not the most pertinent, as these are different systems operating under different constraints, and will continually evolve. More relevant is the potential efficiency gain offered by FPGA designs demonstrated in Figure 12. The plot implies that an FPGA implementation is more efficient when there is enough load on it. More importantly, there is no point in utilising more FPGA resources, i.e., the FPGA 4E, for a single operation if the load does not saturate a smaller version, i.e., the FPGA 2E.

5.4.3 Resource utilisation. Table 3 in Appendix shows the resources consumed by the different FPGA designs, as well as the resulting maximum clock frequency, and the total power consumption.

[4] https://www.computeexpresslink.org

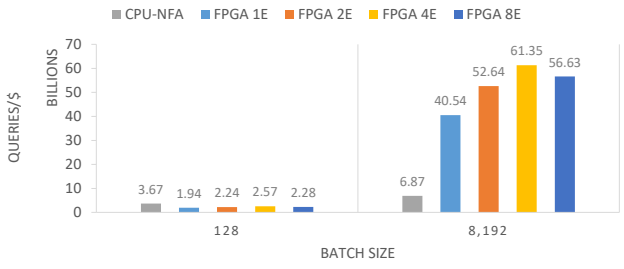


Figure 12: Computing cost for CPU and FPGA deployments in the cloud in billions of queries per USD.

6 DISCUSSION

There are several ways to take advantage of the design just described as a deployment solution within the Amadeus flight search engine. We discuss them here to illustrate the interplay between hardware acceleration and existing systems.

By running MCT on an FPGA, we could free up to 40% of CPU time in the Domain Explorer, which could then, theoretically, accept 40% more requests. However, even the higher number of requests would barely saturate the FPGA-based engine, given that it provides performance that is orders of magnitude higher than the current CPU implementation. While it is not possible to choose a different board on Amazon

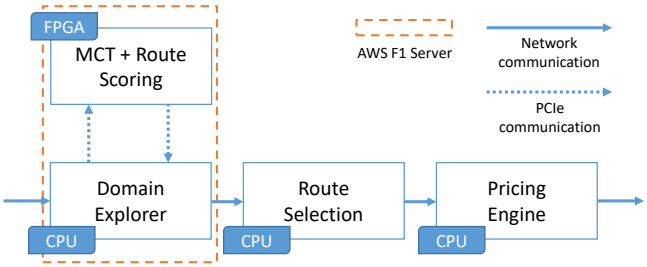


Figure 13: Moving Route Scoring [21] and MCT to an FPGA running on the same machine as the Domain Explorer.

cloud, as of March 2020, a smaller FPGA fitting fewer NFA-EEs could be used for a deployment on-premises. It would drastically reduce the cost of the FPGA, while ensuring the same quality of service with 40% less servers; reducing from 400 CPU-only Domain Explorers to about 240 CPU-FPGA nodes.

Alternatively, the extra capacity of a big FPGA board could be used to combine the MCT module with Route Scoring, another part of the flight search engine successfully accelerated using FPGAs [21]. The Route Scoring would move earlier in the flow, directly inside the Domain Explorer, to score the routes during the flight domain exploration (Figure 13). By doing so, the Route Scoring would be able to process several tens of thousands of routes in the Domain Explorer, instead of only few hundreds inside the Route Selection, while respecting the same response time constraint. Both MCT and Route Scoring would then be on the same FPGA, and pipelined together to minimise the back and forth with the CPU. Such pipeline would require an additional refactoring of the Domain Explorer, since the notions of route for MCT and Route Scoring are not exactly the same, but such deployment would optimise the FPGA occupancy and its usage, while combining the potential business benefits of both modules.

In terms of the resulting system, what we propose is narrower than a general purpose business rule engine. A good analogy is a key value store (KVS) versus a full relational database engine. On one hand, a KVS is faster and more scalable for a narrower set of use cases and operations. On the other hand, a database supports a far wider set of use cases, but the price of generality is reduced performance and elasticity. One way to look at what we propose here is to understand it as a form of "key value store" for business rules that significantly improves performance and resource utilisation for a narrower set of use cases. As with KVS, the advantages of a specialised system are significant enough to be considered a replacement to more general solutions.

7 CONCLUSIONS

In the context of a commercial flight search engine, in this paper we have explored how to use hardware acceleration through FPGAs to implement a business rule management system capable of sustaining strict throughput and latency requirements. We described how to efficiently model large collections of business rules using NFAs. The result is available as open-source[5].

Experimental results demonstrate a performance gain of several orders of magnitude when compared with the current implementation, Drools, and a CPU version of the design. The results also demonstrate the feasibility of running the system in the cloud. In addition, the design is suitable to be deployed on FPGAs embedded as co-processors, which opens up the possibility of reducing the number of CPUs needed to address the computing demand of the flight search engine.

The current results only cover rule matching without any forward or backward chaining. As part of future work, we will explore incorporating richer semantics on the rule attributes, as well as a more general rule resolution algorithm. On the hardware acceleration side, we intend to explore other architectures, including smart-NICs, clusters of FPGAs, and FPGAs with High Bandwidth Memory.

ACKNOWLEDGMENTS

We would like to thank Xilinx for the generous donations of the Alveo boards. Part of the work of Fabio Maschi and Muhsen Owaida was funded by a grant from Amadeus.

REFERENCES

[1] Michela Becchi and Patrick Crowley. 2007. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2007, New York, NY, USA, December 10-13, 2007*, Jim Kurose and Henning Schulzrinne (Eds.). ACM, 1. https://doi.org/10.1145/1364654.1364656

[2] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. 2003. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.* 28, 4 (2003), 467–516. https://doi.org/10.1145/958942.958947

[3] Yanlei Diao and Michael J. Franklin. 2003. High-Performance XML Filtering: An Overview of YFilter. *IEEE Data Eng. Bull.* 26, 1 (2003), 41–48. http://sites.computer.org/debull/A03mar/yfilter.ps

[4] Drools. 2019. *Drools Business Rule Management System.* Retrieved November 19, 2019 from https://www.drools.org/

[5] Yuanwei Fang, Tung Thanh Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, Milos Prvulovic (Ed.). ACM, 533–545. https://doi.org/10.1145/2830772.2830809

[5] https://www.systems.ethz.ch/fpga

[6] Yuanwei Fang, Chen Zou, and Andrew A. Chien. 2019. Accelerating Raw Data Analysis with the ACCORDA Software and Hardware Architecture. *PVLDB* 12, 11 (2019), 1568–1582. https://doi.org/10.14778/3342263.3342634

[7] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[8] Charles Forgy. 1982. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.* 19, 1 (1982), 17–37. https://doi.org/10.1016/0004-3702(82)90020-0

[9] Thilan Ganegedara, Yi-Hua E. Yang, and Viktor K. Prasanna. 2010. Automation Framework for Large-Scale Regular Expression Matching on FPGA. In *International Conference on Field Programmable Logic and Applications, FPL 2010, August 31 2010 - September 2, 2010, Milano, Italy*. IEEE Computer Society, 50–55. https://doi.org/10.1109/FPL.2010.21

[10] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 44:1–44:12. https://doi.org/10.1109/MICRO.2016.7783747

[11] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2004. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.* 29, 4 (2004), 752–788. https://doi.org/10.1145/1042046.1042051

[12] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. 2013. A Catalog of Stream Processing Optimizations. *Comput. Surveys* 46, 4 (2013), 46:1–46:34. https://doi.org/10.1145/2528412

[13] IBM. 2012. *IBM Operational Decision Manager.* Retrieved November 19, 2019 from https://www.ibm.com/automation/business-rules

[14] Ilya Kolchinsky and Assaf Schuster. 2019. Real-Time Multi-Pattern Detection over Event Streams. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 589–606. https://doi.org/10.1145/3299869.3319869

[15] Gianfranco Lamperti and Michele Scandale. 2013. Incremental Determinization and Minimization of Finite Acyclic Automata. In *IEEE International Conference on Systems, Man, and Cybernetics, Manchester, SMC 2013, United Kingdom, October 13-16, 2013*. IEEE, 2250–2257. https://doi.org/10.1109/SMC.2013.385

[16] Alessandra Loro, Anja Gruenheid, Donald Kossmann, Damien Profeta, and Philippe Beaudequin. 2015. Indexing and Selecting Hierarchical Business Logic. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1656–1667. https://doi.org/10.14778/2824032.2824064

[17] Microsoft. 2017. *BizTalk Business Rule Engine.* Retrieved November 19, 2019 from https://docs.microsoft.com/en-us/biztalk/core/business-rules-engine

[18] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP?. In *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, William D. Gropp, Pete Beckman, Zhiyuan Li, and Francisco J. Cazorla (Eds.). ACM, 1:1–1:11. https://doi.org/10.1145/3079079.3079100

[19] Eriko Nurvitadhi, Jeffrey J. Cook, Asit K. Mishra, Debbie Marr, Kevin Nealis, Philip Colangelo, Andrew C. Ling, Davor Capalija, Utku Aydonat, Aravind Dasu, and Sergey Shumarayev. 2018. In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 106–110. https://doi.org/10.1109/FPL.2018.00027

[20] Oracle. 2008. *Oracle Business Rules.* Retrieved November 19, 2019 from https://www.oracle.com/technetwork/middleware/business-rules/overview/index.html

[21] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. 2019. Lowering the Latency of Data Processing Pipelines Through FPGA based Hardware Acceleration. *PVLDB* 13, 1 (2019), 71–85. https://doi.org/10.14778/3357377.3357383

[22] Bill Parducci, Hal Lockhart, and Erik Rissanen. 2013. Extensible Access Control Markup Language (XACML), Version 3.0. *OASIS Standard* (2013), 1–154.

[23] Mauricio Salatino, Mariano De Maio, and Esteban Aliverti. 2016. *Mastering jboss drools 6.* Packt Publishing Ltd.

[24] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 403–415. https://doi.org/10.1145/3035918.3035954

[25] Jens Teubner and Louis Woods. 2013. *Data Processing on FPGAs: Synthesis Lectures on Data Management.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00514ED1V01Y201306DTM035

[26] Louis Woods, Jens Teubner, and Gustavo Alonso. 2010. Complex Event Detection at Wire Speed with FPGAs. *PVLDB* 3, 1 (2010), 660–669. https://doi.org/10.14778/1920841.1920926

[27] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 407–418. https://doi.org/10.1145/1142473.1142520

[28] Xilinx. 2016. *UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices.* Technical Report. https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[29] Xilinx. 2019. *Improve Your Data Center Performance With The New QDMA Shell.* Retrieved November 20, 2019 from https://forums.xilinx.com/t5/Adaptable-Advantage-Blog/Improve-Your-Data-Center-Performance-With-The-New-QDMA-Shell/ba-p/990371

[30] Xilinx. 2020. *Vitis Unified Software Platform.* Retrieved February 18, 2020 from https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[31] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. 2008. Compact architecture for high-throughput regular expression matching on FPGA. In *Proceedings of the 2008 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2008, San Jose, California, USA, November 6-7, 2008*, Mark A. Franklin, Dhabaleswar K. Panda, and Dimitrios Stiliadis (Eds.). ACM, 30–39. https://doi.org/10.1145/1477942.1477948

[32] Yi-Hua Edward Yang and Viktor K. Prasanna. 2009. Software Toolchain for Large-Scale RE-NFA Construction on FPGA. *Int. J. Reconfig. Comp.* 2009 (2009), 301512:1–301512:10. https://doi.org/10.1155/2009/301512

A APPENDIX

Table 2: Platforms used in the experimental evaluation.

	CPU	Cores	RAM	FPGA	PCIe Bandwidth	($/Hr)
AWS Instance						
CPU c5.12xlarge	Intel Xeon Platinum 8275CL	48	92 GB	–	–	2.30
FPGA f1.2xlarge	Intel Xeon E5-2686 v4	4	122 GB	UltraScale+ VCU9P, 270 Mb	10 GB/s	1.82
On-Premises						
Benchmarks	Intel Xeon E5-4650 v2	40	512 GB	–	–	
Drools server	Intel Xeon E5-2640 v3	16	125 GB	–	–	
FPGA server	Intel Xeon Gold 6234	8	157 GB	Xilinx Alveo U250, 360 Mb	11.3 GB/s	

Table 3: Resource utilisation, maximum clock frequency and power consumption on different FPGA boards and number of NFA-EEs. Shell's consumption is also reported.

		LUTs		Registers		BlockRAM		UltraRAM		Clock	Power
U250 QDMA	Available	1,728,000	(100%)	3,456,000	(100%)	2,688	(100%)	1,280	(100%)		3.32 W
	Shell	169,315	(9.80%)	210,473	(6.09%)	177	(6.58%)	5	(0.39%)	300 Mhz	
	H2Desc 1E	52,908	(3.06%)	76,640	(2.22%)	30	(1.12%)	100	(7.81%)	500 Mhz	10.72 W
	H2Desc 2E	60,237	(3.49%)	85,852	(2.48%)	30	(1.12%)	200	(15.63%)	413 Mhz	11.53 W
	H2Desc 4E	74,759	(4.33%)	105,009	(3.04%)	30	(1.12%)	400	(31.25%)	292 Mhz	13.28 W
	H2Desc 8E	104,282	(6.03%)	143,350	(4.15%)	30	(1.12%)	800	(62.50%)	162 Mhz	16.93 W
	H2Desc AE	119,053	(6.89%)	161,911	(4.68%)	30	(1.12%)	1,000	(78.13%)	168 Mhz	18.75 W
U250 XDMA	Available	1,728,000	(100%)	3,456,000	(100%)	2,688	(100%)	1,280	(100%)		3.32 W
	Shell	104,112	(6.03%)	160,859	(4.65%)	165	(6.14%)	-	(0.00%)	300 MHz	
	H2Desc 1E	47,779	(2.76%)	80,637	(2.33%)	78	(2.90%)	100	(7.81%)	201 MHz	13.18 W
	H2Desc 2E	55,139	(3.19%)	90,021	(2.60%)	78	(2.90%)	200	(15.63%)	215 MHz	15.04 W
	H2Desc 4E	69,798	(4.04%)	107,968	(3.12%)	78	(2.90%)	400	(31.25%)	215 MHz	18.45 W
	H2Desc 8E	99,126	(5.74%)	146,126	(4.23%)	78	(2.90%)	800	(62.50%)	190 MHz	25.56 W
	H2Desc AE	113,786	(6.58%)	166,479	(4.82%)	78	(2.90%)	1,000	(78.13%)	149 MHz	29.35 W
AWS F1	Available	1,181,768	(100%)	2,363,536	(100%)	2,160	(100%)	960	(100%)		3.39 W
	Shell	157,748	(13.35%)	205,917	(8.71%)	199	(9.21%)	43	(4.48%)	250 MHz	
	H2Desc 1E	28,903	(2.45%)	42,258	(1.79%)	31	(1.44%)	100	(10.42%)	402 MHz	35.60 W
	H2Desc 2E	36,419	(3.08%)	51,444	(2.18%)	31	(1.44%)	200	(20.83%)	388 MHz	39.97 W
	H2Desc 4E	50,898	(4.31%)	70,668	(2.99%)	31	(1.44%)	400	(41.67%)	282 MHz	41.56 W
	H2Desc 8E	80,308	(6.80%)	108,970	(4.61%)	31	(1.44%)	800	(83.33%)	191 MHz	49.76 W