Do Arduinos dream of efficient reasoners?

Alexandre Bento, Lionel Médini, Kamal Singh, Frederique Laforest

Do Arduinos dream of efficient reasoners?

Alexandre Bento[1], Lionel Médini[2], Kamal Singh[3], and Frédérique Laforest[1]

[1] LIRIS UMR CNRS 5205, INSA Lyon, Villeurbanne, France
[2] LIRIS UMR CNRS 5205, Université Claude Bernard Lyon 1, Villeurbanne, France
[3] LaHC UMR CNRS 5516, Université Jean Monnet, Saint-Étienne, France
{alexandre.bento, lionel.medini, frederique.laforest}@liris.cnrs.fr
kamal.singh@univ-st-etienne.fr

Abstract. The Semantic Web of Things enhances the Internet of Things with Web technologies as well as Knowledge Graphs and reasoning. Traditional reasoners are too heavy in terms of memory footprint and/or processing time to be implementable on things. In this work, we present LiRoT, a lightweight incremental reasoner that can be embedded in constrained objects, so that reasoning on them in a fog architecture becomes possible. The focus of this work is to reduce drastically memory footprint while paying attention to processing time, hence usual optimization techniques are not fully adequate. We provide evaluations that (i) compare our system to the state of the art and (ii) show the effective benefits of the different optimizations we have implemented.

Keywords: Semantic Web · Reasoning · Web of Things · Embedded systems · Optimization

1 Introduction

Today, more and more applications require a connection to the physical world to capture information from the environment as well as to act on it. The Internet of Things provides answers to such needs by connecting sensors and actuators to computers; the Web of Things (WoT) intends to do it using the Web standards; and the Semantic Web of Things (SWoT) enhances the WoT by adding the expressive power of Knowledge Graphs as well as reasoning capabilities. At the same time, the huge number of things and the total volume of produced data has raised the need for distributed edge and fog architectures [22] where data are processed as close as possible to their production and consumption locations. Fog computing architectures can involve different types of nodes. Some of these fog nodes might have limited amounts of energy or/and bandwidth. Reasoning on devices with limited resources, such as microcontroller-based[4] ones, will be an important enabler for reasoning in such architectures. It is obvious that very small microcontrollers with a few kilobytes of RAM will not be able to process

[4] Microcontrollers are small processing units designed to run embedded applications, in contrast to more powerful microprocessors that can execute general purpose applications.

highly expressive reasoning tasks about very large datasets, and that enabling semantic reasoning on constrained devices is a matter of trade-offs. Nevertheless, significantly useful tasks such as classification using subsets of RDF-S[5] can be performed on devices with around 100 KB RAM and 100 MHz clock speed. However, modern state-of-the-art reasoners are optimized for speed and high data volumes and many of the optimizations on which they rely (e.g. exploiting multiple cores and highly parallel architectures) are not usable on such devices, so that finally none of them seems to fit for edge/fog reasoning.

Focusing on typical SWoT use cases on such architectures, the application, ruleset and ontology are usually known in advance. Hence, they can be flashed on the device (we herein assume there is enough room for that). Whilst running the application, sensor data that arrives periodically should be processed on the fly. Given our fog architecture assumption, we focus on a scenario where small data quantities issued by one or a few sensors require to update the reasoner's internal state. For example, in the CoSWoT[6] project, we target applications such as field watering or frost prevention that are based on temperature and humidity data provided for each field a couple of times each day, and that can rely on decisions made locally for each field (more details in our use case[7]). W3C WoT use cases also highlight connectivity and autonomy constraints in edge architectures[8], making reasoning on edge nodes relevant. We herein choose to process data incrementally [14] rather than as stream [4], to avoid the need for the reasoner to handle time or data windows.

In this work, we present LiRoT, a lightweight incremental reasoner that can be embedded in resource-constrained nodes of fog architectures. Starting from the well known RETE algorithm, we propose specific improvements that target the above SWoT use cases. The focus of our work is thus mainly on memory frugality, while also considering algorithmic optimization.

The paper is organized as follows. Section 2 reviews previous works on incremental reasoning optimizations and discusses their adequacy to SWoT; it also describes the classic RETE algorithm. Section 3 presents the proposed optimizations for LiRoT, a SWoT-compliant RETE-based reasoner. Section 4 presents two sets of evaluations, (i) to compare our reasoner to state of the art systems and (ii) to show the effects of the implemented optimizations. Section 5 discusses the results and describes optimizations that were experimented but did not improve performance. Section 6 concludes and sketches future directions for our work.

2 Related work

In this section, we review the different works that can be applied to perform reasoning tasks with a focus on small devices. Given the strongly constrained envi-

[5] https://www.w3.org/TR/rdf-mt/#rdfs_entailment
[6] https://coswot.gitlab.io/
[7] https://www.w3.org/TR/wot-usecases/#Agricultural-irrigation
[8] https://www.w3.org/TR/wot-usecases/#edge-computing

ronments on which we intend to deploy our work, we herein focus on lightweight reasoning algorithms, namely rule-based ones. The OWL 2 RL profile[9] has been designed to foster the use of rule-based reasoners. The foundations of how and why to combine rules with ontologies for the Semantic Web are addressed in Description Logic Programs (DLP) [8], which bridge the gap between knowledge representation (KR) and in particular DL and LP[10]. Moreover, in order to allow their deployment on constrained environments, we herein restrict to subsets of the OWL 2 RL ruleset and the RDF-S entailment list[11].

2.1 The RETE algorithm

A rule-based reasoner applies rules to the *facts*[12] contained in a knowledge base (KB) to produce new knowledge, and loops over the set of initial and produced knowledge until no new knowledge is issued. In the reasoning field, the facts that were already present in the KB at the beginning of this loop are called *explicit facts*, and those that have been inferred during the loop are called *implicit facts*.

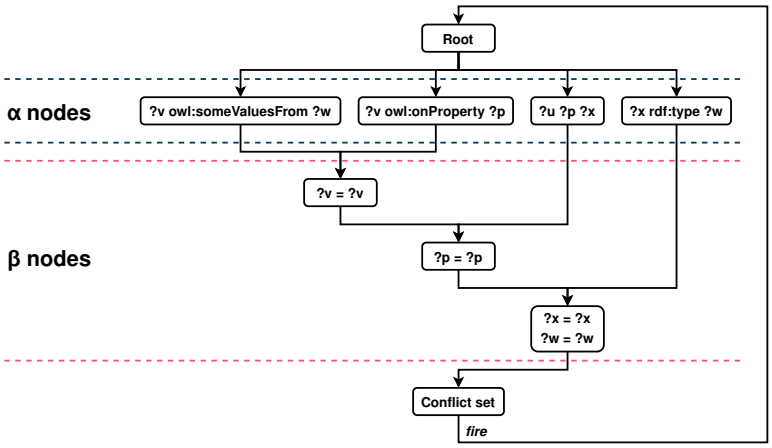

Fig. 1: Example of a RETE network for rule `cls-svf1` from the OWL 2 RL profile: ($?v$ owl:someValuesFrom $?w$) $\wedge$ ($?v$ owl:onProperty $?p$) $\wedge$ ($?u$ $?p$ $?x$) $\wedge$ ($?x$ rdf:type $?w$) $\rightarrow$ ($?u$ rdf:type $?v$)

The RETE [6] algorithm is one of the most well-known algorithms to process rulesets over a KB. Rules are represented with a trie structure called the RETE network. A RETE network is composed of two main layers (see example on Figure 1):

[9] https://www.w3.org/TR/owl2-profiles/#OWL_2_RL
[10] https://en.wikipedia.org/wiki/Logic_programming
[11] https://www.w3.org/TR/rdf11-mt/#entailment-rules-informative
[12] As we herein consider the KB as being an ontology expressed in OWL 2 RL under RDF-based semantics, facts are RDF triples.

- The **alpha nodes**: each alpha node is associated with an atomic condition in a rule (e.g. ?x rdf:type foaf:Person), and performs a match operation over the whole knowledge base (i.e. explicit and implicit facts). Alpha nodes contain *alpha memories*, which store the facts that match the node's condition. An alpha node is related to a single rule, even if a same condition is shared by multiple rules. An alpha node has a single output edge, that leads to a beta node.
- The **beta nodes**: beta nodes perform join operations between two nodes. Beta nodes can be placed either after two alpha nodes, or after a beta and an alpha node if the associated rule has more than two conditions (then join operations are performed sequentially). Beta nodes contain *beta memories*, that store variable substitutions that are compatible with the node's parents. A beta node has exactly two input edges and one output edge.

2.2 Incremental reasoning

SWoT use cases imply multiple data insertion and deletion steps to reflect the state of dynamically changing physical environments. When data evolves over time, explicit facts may need to be inserted in and deleted from a reasoner. Implicit facts derived from them also need to be updated. To tackle this issue, incremental maintenance allows insertion and deletion of explicit facts without re-performing the materialization operation from scratch.

The RETE algorithm natively supports incremental maintenance using two adaptations. Incremental insertion is managed by splitting alpha memories into two parts: new facts and already-processed facts. Deletion is managed by adding a data structure inside alpha and beta nodes to connect implicit facts with the explicit facts they come from. This is traditionally done using lists, although a tree structure is also possible [5].

Jena [3] is a reference rule-based OWL reasoner based on RETE that implements the DL subset of the first OWL specification. It provides an easy-to-use API for Java programmers and has been widely used as such. However, all constrained objects do not support this language, and especially a garbage-collecting function, so it is out of scope for the field of SWoT and embedded reasoning.

CLIPS[13] [12] is a widely used expert system tool. It uses a complete object-oriented language for writing expert systems. It employs the RETE algorithm for reasoning. However, RDF triples are not supported natively by CLIPS. Other tools, such as R-DEVICE [1], are needed to import RDF into CLIPS.

The Delete/Rederive (DRed) algorithm [10] handles deletion. It first over-deletes all implicit facts that depend on the deleted facts. Then it rederives the implicit facts that can be inferred another way. Rederivation is iteratively applied. Some other works use a variant of the DRed algorithm, for example in [19].

[13] http://www.clipsrules.net/

For supporting incremental reasoning, RDFox [13] uses a backward-forward algorithm. Unlike DRed that searches after over-deletion, the backward-forward algorithm uses an approach that first searches for the alternative derivations. This is done by using a combination of backward and forward chaining.The induced performance gain is particularly visible with implicit facts that are derived from numerous chained deductions (e.g. rdfs:subClassOf). GraphDB[14] uses the same approach. RDFox has a high memory footprint (especially due to a high number of indexes) and is optimized for architectures that allow parallel computing, hence it is not designed to be embedded in constrained devices.

HyLAR+ [17] is an incremental reasoner implemented in JavaScript and targeting Web applications, which has been improved with a so-called "tag-based" approach [18], that allows for fastly performing multiple fact insertion/deletion. This approach is inspired from that initiated by [7] on improving reasoning about evolving versions of ontologies. The general idea is to keep trace of previous reasoning computations originated by changes in the graph, in order to respond more quickly to similar changes in the future. The drawback of this approach is that it requires to store history as extra information, which is therefore not suitable for memory-constrained devices.

2.3 Embedded Reasoning

Many existing semantic reasoners are too resource-intensive to be directly ported on resource-constrained devices such as objects or sensors. Only a few works embed reasoning in constrained devices. Some of them are designed for mobile phones and not for more constrained devices. For example, [2] studied porting Description Logics (DL) reasoners on mobile Android-based devices. It is worth noting that smartphones have much higher computational capabilities as compared to the devices we target.

An OWL reasoner for embedded devices was proposed in [15], it is based on CLIPS. They considered OWL 2 RL. Their system was implemented and tested on Gumstix Verdex Pro which has 400 MHz CPU, 64 MB RAM, and 16 MB Flash. Note that in our work, we are targeting embedded devices that have several orders of magnitude less RAM (around 500 KB).

$\text{RETE}_{\text{pool}}$ [20] is a RETE-based reasoner that aims to reduce its memory footprint by reducing data duplication during rule based reasoning, while specifically considering OWL 2 RL. To do so, it uses one shared memory for all alpha nodes in the network. This way, duplicates are eliminated during insertion. In cases where a RDF store is used along the reasoner, another level of duplication is removed by using this store directly as the RETE memory, each alpha node having references to triples contained in the store. This saves memory, at the cost of speed degradation. Their experiments were conducted on smartphones and laptops.

[14] https://graphdb.ontotext.com/documentation/free/reasoning.html#retraction-of-assertions

Another work based on the RETE algorithm is called COROR [16]. It uses the following composition algorithms to reduce memory consumption. It selectively loads only the rules that are required, by creating a rule-construct dependencies set. Next it decomposes the RETE algorithm in two phases. The first phase does an initial matching. Then the next phase builds the next part of the RETE network using statistics collected from the first phase, that allow to reorder rules and conditions according to their selectivity (the most selective conditions are matched first; this optimization is well-known in database management systems, for ordering join operations). The first cycle is then completed by joining the facts obtained from the first phase. Results show that COROR reduces the memory footprint by 74% on average. COROR also uses rewriting for rules that are known to be resource–and time–consuming, such as rules involving owl:sameAs and wildcard conditions. COROR was experimented on a SunSPOT platform, which has a similar memory size to our target platforms but uses the Java language. These experiments showed very low speed: they reported 1561 seconds to process the WINE ontology that contains 1833 triples with the ρD* ruleset.

This state of the art has shown that the proposed reasoners of the literature are not well suited to be run on microcontroller-based platforms. Nevertheless, they include some optimization proposals that target memory footprint. The next section will present how we obtained a reasoner that can be run on platforms of the Arduino family, by extracting the best ideas of the literature and adapting them to this specific context.

3 LiRoT: improving RETE for the SWoT

We propose optimizations over the traditional RETE algorithm. These optimizations are built with the objective to be embeddable on platforms like Arduino or ESP32 architectures. So they are focused on memory footprint, as it is a strict criterion for such platforms. We also pay attention to processing time, checking that it remains within acceptable bounds.

3.1 Term indexing

To save memory, we use an index over the terms within the reasoner. Various implementations exist in the literature, relying on data structures such as linked lists, arrays or binary search trees. A hash table can also be used to do this in an efficient manner: indeed, on average, a hash table has a linear space complexity, and constant time complexity for insertion, search and deletion, which is better than the previously cited data structures. Hence we chose this option to make a term index.

3.2 Merging alpha memories

Alpha memories have already been the place for optimization of the RETE algorithm, like in $\text{RETE}_{\text{pool}}$ where all alpha memories are merged into one common

memory. Here, to optimize smartly the RETE algorithm memory footprint, we merge memories of alpha nodes that share a syntactically similar rule condition.

Let $c_1 = (s_1, p_1, o_1)$ and $c_2 = (s_2, p_2, o_2)$ be two conditions. c_1 and c_2 are similar if either:

- s_1 and s_2 (resp. p_1 and p_2, o_1 and o_2) are variable terms
- s_1 and s_2 (resp. p_1 and p_2, o_1 and o_2) are not variable terms and $s_1 = s_2$ (resp. $p_1 = p_2$, $o_1 = o_2$)

As we merge memories of all alpha nodes that share similar rule conditions, each rule condition is checked by only one node. This optimization saves both memory and time. This is particularly useful in rulesets like RDFS, where multiple rules use a wildcard condition (a condition that matches all facts in the KB, e.g. $?x$ $?p$ $?y$).

This optimization implies that each alpha memory is potentially connected to multiple beta nodes (one for each original alpha memory that has been merged), which is not the case in the original RETE network. It has an impact on the management of the alpha memories, as a fact remains new as long as one of its beta nodes has not yet processed it (see section 2.2). Triggering the execution of the beta nodes connected to this alpha memory is thus required before pushing new facts in the already-processed part of the alpha memory.

3.3 Optimizing incremental maintenance

To handle incremental maintenance, we have used a similar philosophy as that of the backward-forward algorithm [13]. Although not initially designed to work with RETE, it is compatible with its network structure: each terminal beta node (i.e. the last beta node of a rule) stores a list of the implicit facts that it has produced. Every implicit fact contains the list of beta memories that led to its production. When an explicit fact is removed from the knowledge base, it is first removed from the memory of its matching alpha nodes, then the corresponding variable substitutions are removed from beta memories following the same route as when adding a new explicit fact. After this step, if an implicit fact has no more cause coming from the beta node that produced it, the algorithm first searches through all other terminal beta nodes if the same implicit fact was produced somewhere else. If not, it is deleted from the knowledge base. This avoids unnecessary deletions, in case an implicit fact was obtained in multiple ways.

3.4 Implementation details

LiRoT is written in C: low-level languages are more suited for constrained platforms because they allow for more fine-grained memory management. Rust[11] could also have been an option, but it is relatively new and not yet available on most platforms.

LiRoT is composed of two modules:

- The core algorithm, based on RETE and including the proposed optimizations. It currently uses Sord[15], a C library providing a lightweight in-memory triplestore, to implement and store RDF triples.
- A wrapper to the core algorithm providing an RDF-JS-like API[16] to query the reasoner. It relies on Serd[17] to parse and serialize RDF triples in Turtle, N-Triples, N-Quads and TriG formats.

In the core algorithm, to implement the term index, we use the efficient hashtable implementation from the uthash[18] library. To compute terms hashes, we use the hash function provided by Sord.

LiRoT comes with two versions: a Linux version and an Arduino version. It was tested on Manjaro Linux, Arduino Due and ESP-32 platforms.

LiRoT source code is available at https://gitlab.com/coswot/lirot .

4 Evaluation and results

4.1 Dataset and evaluation method

We used the LUBM benchmark[9] to generate 12 synthetic datasets of various sizes in the domain of universities. These datasets contain from 0 to 10,000 explicit triples representing assertions, in addition to the ontology itself (293 triples).

We have implemented the three rulesets RDFS-Simple, RDFS-Default and RDFS-Full (provided by Apache Jena[19]) for each tested reasoner. We provide the lists of rules in the LiRoT source code repository.

To compare our approach to other incremental reasoners and to assess the effectiveness of different versions of our algorithm, we have run four types of experiments on each of the 12 datasets:

- **full materialization**. The dataset is entirely loaded at once, then the reasoning is launched.
- **incremental insertions**. We randomly split each dataset into two equal parts. The first half is inserted into the reasoner. The second half is divided into five fragments, each representing 10% of the whole dataset, and are added sequentially.

[15] https://github.com/drobilla/sord
[16] https://rdf.js.org/
[17] https://github.com/drobilla/serd
[18] https://github.com/troydhanson/uthash
[19] Rulesets are described at https://jena.apache.org/documentation/inference/#RDFSconfiguration

- **incremental deletions**. The idea here is to delete multiple parts of a preloaded dataset. We first load the whole dataset, execute an initial reasoning task, then sequentially remove the same subsets as described before.
- **incremental insertions and deletions**. Here we sequentially add then delete parts of the dataset. The first half of the dataset is loaded, then each subset is first inserted then deleted. This is the closest test to the actual use cases that we performed using LUBM. In the use cases for which this reasoner is designed, the "same" triples will be added and removed (possibly with some variations).

We use two performance metrics: i) maximum resident set size[20] (maxRSS): in the context of constrained objects, memory size is a mandatory limitation, and ii) execution time.

Experimentation materials are available at https://gitlab.com/coswot/lirot-experiments-eswc-2022, where one can find datasets, scripts used to run all reasoners, raw results files and plots.

In the following sections, RETE denotes our baseline implementation of the RETE algorithm; RETE+alpha is RETE with the optimization on alpha memories as described in section 3.2; RETE+terms denotes RETE with the index on terms described in section 3.1; LiRoT is RETE with the optimizations on alpha memories and the index on terms. The optimization on incremental reasoning is present in all versions of RETE, RETE+alpha, RETE+terms and LiRoT.

4.2 Correctness verification

To ensure that LiRoT produces correct results, we compared its output with that of Apache Jena, which is a well tested reasoner. We performed these tests both on the insertion and deletion algorithms, and got the same output in all cases.

4.3 Comparison with other reasoners

We compared our approach to two standard incremental reasoners: Apache Jena and RDFox. To do so, we ran each reasoner on the same desktop computer (MSI GF63 Thin 10SCXR-046FR Dragon Station, with an Intel Core i7-10750H CPU and 32 GB of DDR4 2666 MHz RAM). We forced the execution of each reasoner on only one CPU thread, to mimic the behavior of more constrained devices. We ran each test 20 times, removed the most extreme values (5% lowest and 5% highest) and computed the average maxRSS and execution time for each reasoner and dataset.

Figure 2 compares LiRoT with Jena and RDFox for the easiest type of test (full materialization with the RDFS-Simple ruleset) and the most difficult one (successive insertions and deletions with the RDFS-Full ruleset).
All other results lie in between these two extreme cases; figures for the other tests are available online; they show similar trends to these figures.

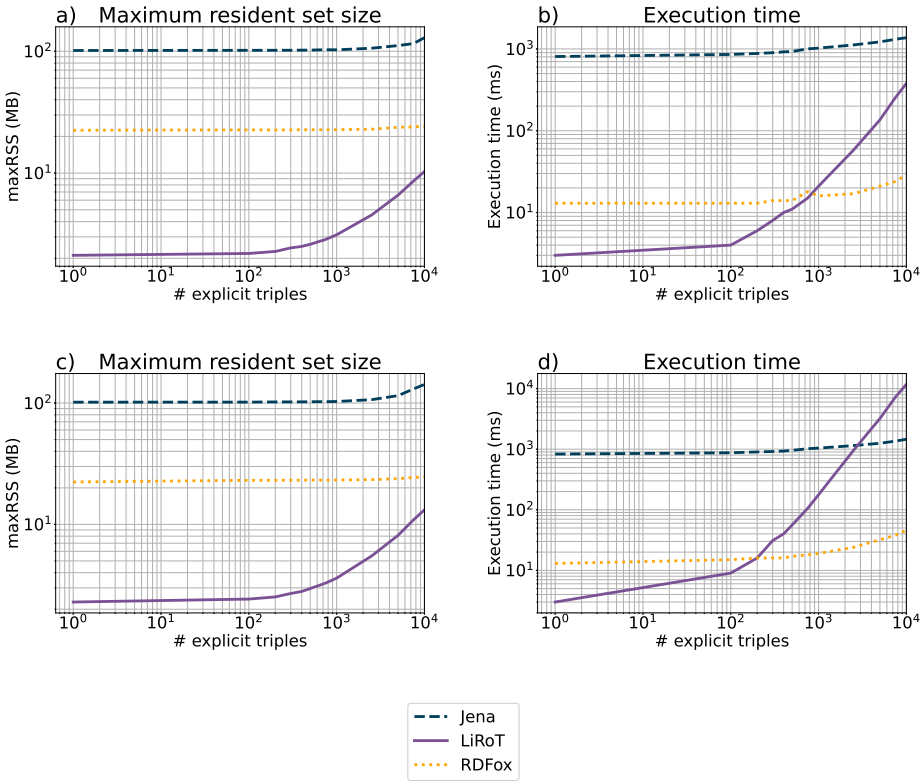

Fig. 2: maxRSS and execution time comparisons between LiRoT, Jena and RDFox.
Figures a) and b) show a full materialization using the RDFS-Simple ruleset. Figures c) and d) show successive insertions and deletions using the RDFS-Full ruleset.

With the RDFS-Simple ruleset and a full materialization, LiRoT uses 58-91% less memory than RDFox, and 92-98% less than Jena. With the RDFS-Full ruleset and incremental insertions and deletions, LiRoT uses 46-90% less memory than RDFox, and 91-98% less than Jena.
With RDFS-Simple and a full materialization, LiRoT is 72-93% faster than Jena. It is faster than RDFox for datasets under 1000 explicit facts and slower for larger datasets. With RDFS-Full and incremental insertions and deletions, LiRoT is faster than Jena (resp. RDFox) for datasets under 2500 (resp. 200) facts. Other configurations have maxRSS and execution time values in between these intervals.

[20] The maximum amount of RAM used by a program throughout its execution.

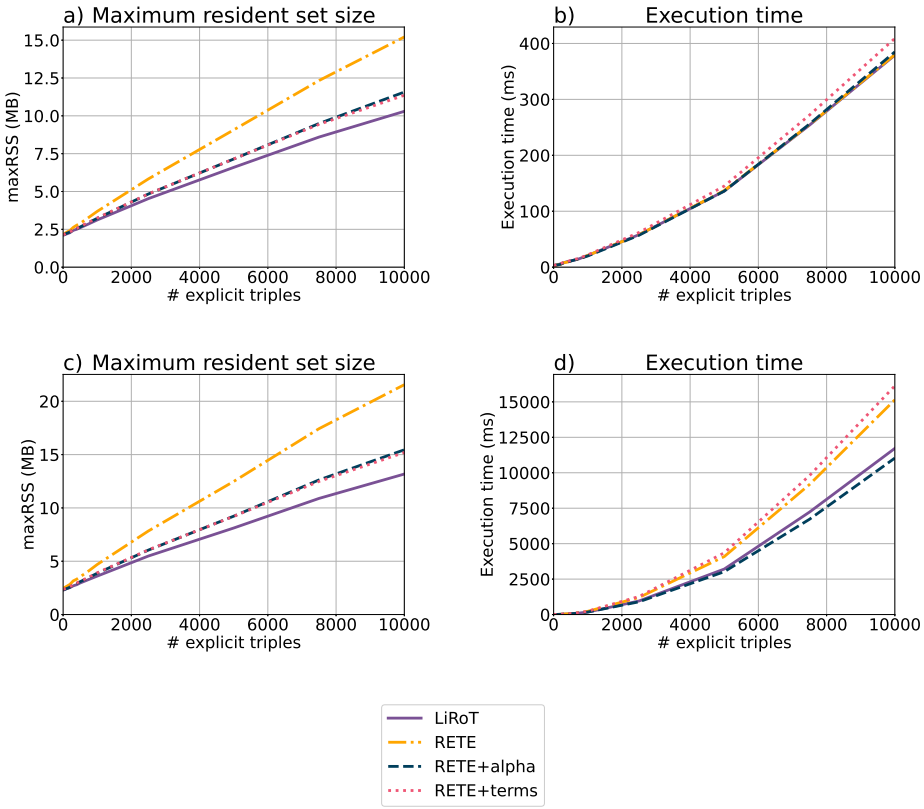

Fig. 3: maxRSS and execution time comparisons between different optimizations of the RETE algorithm.
Figures a) and b) show a full materialization using the RDFS-Simple ruleset.
Figures c) and d) show successive insertions and deletions using the RDFS-Full ruleset.

4.4 Improvements of the RETE Algorithm

On the same desktop setup as previous evaluations, we ran each version of our RETE algorithm baseline and improvements.

Figure 3 shows the different maxRSS and execution times for all four RETE-based algorithms, and in the same configurations as above. With the RDFS-Simple ruleset (resp. RDFS-Full), we find that sharing similar alpha nodes across rules allows to save up to 24% (resp. 28%) memory compared to our baseline RETE implementation. The use of an index on terms saves up to 25% (resp. 29%) memory compared to baseline. The combination of both saves up to 32% (resp. 41%) memory.

Figure 3 shows that for easier types of tests (full materialization with the RDFS-Simple ruleset), optimizations over the RETE baseline have a small im-

pact on the execution time (+1% for the optimization on alpha nodes, +7.6% for the optimization on terms, -0.2% with both optimizations, for the largest dataset; the difference is even smaller for smaller datasets). For the most difficult type of tests (successive insertions and deletions with the RDFS-Full ruleset), optimizations have a more significant impact on the execution time (-37% for the optimization on alpha nodes, +6.5% for the optimization on terms, -23% for the combination of both). The differences among these optimizations are discussed in section 5.2.

4.5 Improvements on embedded devices

We also provide tests for LiRoT on constrained devices, with the RDFS-Simple ruleset. We used an Arduino Due (clock speed of 84 MHz and 96 KB of SRAM) and an Adafruit ESP32 Feather (clock speed of 240 MHz and 520 KB of SRAM). We use incremental insertion of triples to determine the maximum number of triples that can be handled by LiRoT before the device runs out of memory.

On an ESP32 board, we used the standard ESP.getFreeHeap function to measure the memory footprint.

Figure 4 shows the results obtained on ESP32: the baseline RETE implementation is able to load 509 facts (ontology + explicit facts + implicit facts), before the board runs out of memory. With the alpha node-sharing optimization, this number goes up to 656 facts (+22% wrt baseline), 672 facts with the optimization with an index on terms (+24%) and 760 facts with the combination of both optimizations (+49%). All versions have similar execution time for an equal number of facts (for instance, the baseline takes about 200ms to load facts until it runs out of memory).

Platform	Version of RETE	# facts
ESP32	RETE	509
	RETE+alpha	656
	RETE+terms	672
	LiRoT	760
Arduino Due	RETE	240
	RETE+alpha	285
	RETE+terms	290
	LiRoT	349

Table 1: Number of facts (ontology + explicit facts + implicit facts) after reasoning on Arduino Due and ESP32, with the RDFS-Simple ruleset

The hardware architecture of the Arduino Due does not allow to dynamically measure memory usage with standard functions, as we did on the ESP32. To the best of our knowledge, no equivalent library allows to do it on this device. Hence, we measured the maximum number of facts that the reasoner is able to

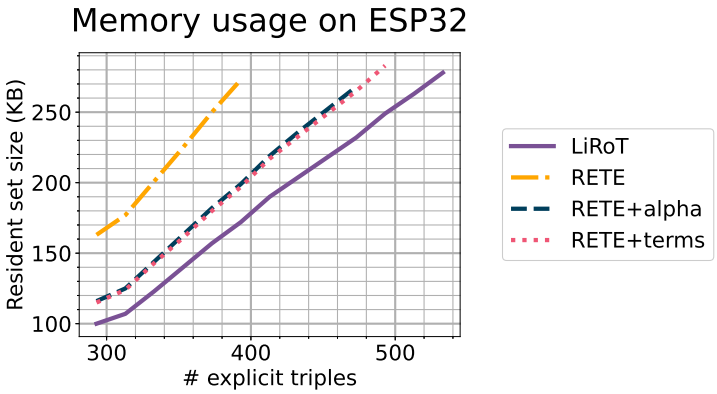

Fig. 4: Comparison of memory usage on a ESP32 board with different optimizations of the RETE algorithm, using the RDFS-Simple ruleset. The horizontal axis shows the number of explicit facts (ontology included).

load and process before the board runs out of memory. Table 1 shows that the optimization on alpha nodes allows to load 19% more facts than the baseline implementation on Arduino Due; the use of an index on terms allows for 21% more facts, and the combination of both allows for 45% more facts, compared to baseline. All versions have similar execution time (about 550ms for the baseline version).

5 Discussion

5.1 Memory usage

On classic PC architectures, LiRoT significantly improves the memory footprint as compared to the literature. Its processing time is lower for small datasets in these settings, but increases rapidly when the number of facts grows. This shows that LiRoT can also be of help on intermediary devices that rely on the same architectures, but have less resources than even smaller computers (e.g. Raspberry Pi).

To the best of our knowledge, LiRoT is the only one able to be actually deployable on Arduino-based platforms[21].

These two points validate our hypothesis of LiRoT as designed for specifically targeting such devices. Processing time is of course to be taken seriously, but must be considered with respect to the findings of Section 4.5.

[21] Indeed, these devices have, in addition to a limited memory size for handling the application data (DRAM), the same kind of limitations for storing the program itself (IRAM). The reasoner should also be compiled specifically for the targeted platform, and use platform-specific available libraries.

5.2 Processing time

Considering the maximum number of facts that a small device can handle, LiRoT remains faster than other reasoners, given that these reasoners could be deployed on the devices. Our experiments also show that the time gained by the optimizations varies according to both sizes of the ruleset and the dataset. It is interesting to note that the LiRoT optimization is not the one that shows the best time performance among all optimizations presented in Section 4.4. However, as they do save memory space, we chose to keep all three optimizations in LiRoT.

Though we admit that restricting the operation range to processing very small amounts of data is not a common goal, it can still benefit to use cases where energy consumption is a critical issue, as well as to embed the reasoner on a relatively more powerful device that has to perform other tasks than reasoning. Indeed, in the CoSWoT project, we intend to include this reasoner as a WoT servient module, so that sensors and actuators can be run autonomously and locally process data as RDF triples.

Moreover, even if RDFox overpasses LiRoT when the number of facts grows, our trade-off assumption seems to remain valid for relatively higher number of facts than those that Arduino-based devices can handle. Our evaluations showed that there is less than 200ms difference for a full materialization of 2000 facts for a whole run of insertions and deletions under the RDFS-Simple ruleset, which sounds to us as acceptable.

The optimizations we implemented for LiRoT come with a few trade-offs. The optimization on alpha nodes could cause concurrent access issues on alpha memories if it were implemented in a multi-threaded application (which LiRoT is not). Term indexing causes a slight loss in performance, as each term insertion requires to check if the term already exists; however, a hash table is a very efficient way to perform this task, so the performance loss is small, in comparison with the overall computation time. Finally, the optimized incremental maintenance algorithm requires to store intermediate reasoning results, which has an impact on memory footprint.

In order to improve the speed of our reasoner, we explored one dead-end that is worth mentioning, and are forseeing two directions:

- One option we tried consists in using an efficient join algorithm in beta nodes. The naive approach, using a nested loop join, reviews all possible matches among two nodes. The sort-merge join algorithm can solve this issue by using sorted data structures in alpha and beta memories and limiting the number of merge steps to the minimum required, to the cost of nodes needing to maintain sorted data structures. However, the relevance of this optimization depends on the ruleset and the dataset sizes: it saves time when the two nodes to join have many elements in their respective memories. This is not the case in our experiments, which showed lower performance.
- The evaluations in which data were to be deleted show higher processing times than expected, while compared to RDFox for instance. Future works

include improving the deletion algorithm by removing intermediary beta node memories, as done in [21].
– Parsing and serializing facts from and to a standard representation format is also time and memory-consuming. Even if Sord and Serd are optimized for this task, they offer many unneeded functionalities that could be removed from the reasoner. In order to diminish both memory usage and processing time, we are currently looking for a more global way to handle compressed RDF data without serialization/deserialization operations at the servient level, relying on binary representations such as CBOR-LD or HDT.

6 Conclusion and future works

We proposed LiRoT, a lightweight incremental reasoner, the first–to the best of our knowledge–that can be embedded on constrained devices such as Arduino Due and ESP32. LiRoT as a tool acts as an enabler for Semantic Web of Things by providing a reasoning capability to constrained devices. Our work also advances the existing state of the art in the fog computing paradigm.

LiRoT implements the RETE algorithm at its heart as the baseline algorithm. Additionally, we studied three optimization schemes to the baseline RETE that resulted in significant memory savings for incremental reasoning. We performed experiments on Linux as well as on embedded systems. We compared the performance of LiRoT with existing reasoners such as RDFox and Jena. As compared to these approaches, our experiments showed that for relatively modest numbers of facts (around 200 to 3000 facts depending on the complexity of the ruleset), which corresponds well to the paradigm of the Semantic Web of Things, LiRoT can do reasoning with lower computation times. LiRoT always had the lowest memory usage for performing reasoning on up to 10,000 facts (maximum number of facts tested). The memory usage was several orders of magnitude lower than RDFox and Jena.

On a desktop configuration, using the LUBM benchmark, our optimizations saved up to 32% of memory with the RDFS-Simple ruleset, up to 36% of memory with RDFS-Default and up to 41% with RDFS-Full. On embedded devices, with the RDFS-Simple ruleset, LiRoT was able to load up to 49% more facts than our baseline RETE implementation.

In the future, we would like to perform tests on other rulesets and to explore more optimization schemes as the ones presented in section 5.2. We will also compare the energy consumption of different optimizations. We would also like to explore distributed and collaborative reasoning algorithms suited to SWoT and embedded environments.

Acknowledgment

This work is supported by grant ANR-19-CE23-0012 from the Agence Nationale de la Recherche, France, for the CoSWoT project.

References

1. Bassiliades, N., Vlahavas, I.: R-device: A deductive rdf rule language. In: International Workshop on Rules and Rule Markup Languages for the Semantic Web. pp. 65–80. Springer (2004)
2. Bobed, C., Yus, R., Bobillo, F., Mena, E.: Semantic reasoning on mobile devices: Do androids dream of efficient reasoners? Journal of Web Semantics **35**, 167–183 (2015)
3. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters. pp. 74–83 (2004)
4. Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook. Data Science **1**(1-2), 59–83 (2017)
5. Doorenbos, R.B.: Production matching for large learning systems. Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE (1995)
6. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. In: Readings in Artificial Intelligence and Databases, pp. 547–559. Elsevier (1989)
7. Grau, B.C., Halaschek-Wiener, C., Kazakov, Y.: History matters: Incremental ontology reasoning using modules. In: The Semantic Web, pp. 183–196. Springer (2007)
8. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: Proceedings of the 12th international conference on World Wide Web. pp. 48–57 (2003)
9. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Journal of Web Semantics **3**(2-3), 158–182 (2005)
10. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. ACM SIGMOD Record **22**(2), 157–166 (1993)
11. Matsakis, N.D., Klock, F.S.: The rust language. ACM SIGAda Ada Letters **34**(3), 103–104 (2014)
12. Meditskos, G., Bassiliades, N.: Clips–owl: A framework for providing object-oriented extensional ontology queries in a production rule engine. Data & Knowledge Engineering **70**(7), 661–681 (2011)
13. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: Rdfox: A highly-scalable rdf store. In: International Semantic Web Conference. pp. 3–20. Springer (2015)
14. Oliya, M., Pung, H.K.: Towards incremental reasoning for context aware systems. In: International Conference on Advances in Computing and Communications. pp. 232–241. Springer (2011)
15. Seitz, C., Schönfelder, R.: Rule-based owl reasoning for specific embedded devices. In: International Semantic Web Conference. pp. 237–252. Springer (2011)
16. Tai, W., Keeney, J., O'Sullivan, D.: Resource-constrained reasoning using a reasoner composition approach. Semantic Web **6**(1), 35–59 (2015)
17. Terdjimi, M., Médini, L., Mrissa, M.: Hylar+ improving hybrid location-agnostic reasoning with incremental rule-based update. In: Proceedings of the 25th International Conference Companion on World Wide Web. pp. 259–262 (2016)
18. Terdjimi, M., Médini, L., Mrissa, M.: Web reasoning using fact tagging. In: Companion Proceedings of the The Web Conference 2018. pp. 1587–1594 (2018)

19. Urbani, J., Margara, A., Jacobs, C., Van Harmelen, F., Bal, H.: Dynamite: Parallel materialization of dynamic rdf data. In: International Semantic Web Conference. pp. 657–672. Springer (2013)
20. Van Woensel, W., Abidi, S.S.R.: Optimizing semantic reasoning on memory-constrained platforms using the rete algorithm. In: European Semantic Web Conference. pp. 682–696. Springer (2018)
21. Wright, I., Marshall, J.A.: The execution kernel of rc++: Rete*, a faster rete with treat as a special case. Int. J. Intell. Games & Simulation **2**(1), 36–48 (2003)
22. Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., Kong, J., Jue, J.P.: All one needs to know about fog computing and related edge computing paradigms: A complete survey. Journal of Systems Architecture **98**, 289–330 (2019)