Benchmarking of triple stores scalability for MPSoC trace analysis

Leon Constantin Fopa
University of Grenoble, LIG
681 rue de la passerelle
St Martin d'Heres, France
fopal@imag.fr

Fabrice Jouanot
University of Grenoble, LIG
681 rue de la passerelle
St Martin d'Heres, France
fabrice.jouanot@imag.fr

Alexandre Termier
University of Grenoble, LIG
681 rue de la passerelle
St Martin d'Heres, France
alexandre.termier@imag.fr

Maurice Tchuente
University of Yaounde 1
LIRIMA and IRD-UMI 209
UMMISCO
BP 337 Yaounde, Cameroon
maurice.tchuente@lirima.org

Oleg Iegorov
STMicroelectronics
850 Rue Jean Monnet
Crolles, France
oleg.iegorov@st.com

ABSTRACT

A Multi Processor System-on-Chip (MPSoC) is a complex embedded system used in consumer electronic devices, such as smartphones, tablets and set-top boxes. In order to cope with the complexity of MPSoC architectures, software developers rely on post-mortem trace analysis for application debugging or optimization. The traces are explored to localize expected and unexpected programs behaviors. However, the low semantic value of low-level trace events make the trace exploration difficult. We propose to perform trace exploration through an ontology which adds semantics to events and provides a declarative language for querying data. Because traces can be huge, such an ontology contains a large number of instances stored as RDF triples. Because analysts need fast results on classical computer, an efficient system for query answering is preferred. Therefore, saturating, loading and querying those triples pose a scalability challenge to state-of-the-art knowledge base repositories (KBR). In this paper, we have conducted a benchmark of 7 KBRs: *Jena*, *Sesame-native*, *Sesame-memory*, *tdb*, *sdb*, *rdf-3x* and *vertical-mdb*, to test their scalability in a non-distributed environment close to analyst environment. We used these KBRs to analyze real traces through VIDECOM, an ontology we designed for trace analysis of applications on MPSoC. Results show that *vertical-mdb* has a loading rate 3 times faster than the others. It is the only KBR able to saturate the biggest trace of our dataset without exceeding system memory and to run complex queries on it in an acceptable time. Other approaches failed, due to memory limitation or inefficient join implementation.

1. INTRODUCTION

Multi Processors Systems-on-Chip (MPSoC) are small chips containing multiple components like processors, memory units, buses, Graphical Processor Unit (GPU), input/output ports. They are widely used in our everyday life through mobile phones, washing machines, automotive control, flight control and set-top boxes. Developing embedded software on MPSoC is difficult because of the inherent parallelism of these chips. Indeed, industrial studies on quality control of embedded softwares indicate high defect densities of 13 major bugs per 1000 lines of code [13]. In multimedia applications, such inefficient code can cause, for example, frozen images or desynchronized images and sound.

The main task in embedded software debugging or optimization is to track bugs or inefficient code manifestations in order to correct them. Inefficient codes and bugs related to parallelism manifest themselves mostly at runtime. Developers, therefore, rely on post-mortem trace analysis methods to debug embedded software [3]. The basic idea is to run the program against specific tests and to explore its execution trace, in order to compare the observed program behavior with expected behavior. The semantics of the trace events, such as relations and constraints between them, are known by the developers, but are not explicit in the trace. Therefore, characterizing program behaviors in a trace is a challenge.

Interpreting events as program behavior is quite similar to data interpretation in the semantic web. The key idea of the semantic web is to propose logical assertions that relate a resource to some concepts in predefined ontologies [2]. Thus, by using a domain ontology for trace analysis, trace exploration can be done through declarative queries whose results will be closer to developer expectations. Because a trace can consist of several million of events for only few minutes of execution, such an ontology will contain a large number of instances stored as RDF triples, which will definitely pose scalability challenges to knowledge base repositories (KBR).

In this paper, we present VIDECOM, an ontology that we have designed for trace analysis of applications on MPSoC. We present a benchmark of 7 KBRs to test their scalability when they are used to saturate, load and query RDF triples obtained from real trace events mapped to classes and properties of VIDECOM. Because the analyst environment is mainly non-distributed, and because we focus on query answering efficiency on a saturated knowledge base, such KBRs have been chosen mainly for their support of query engine and storing system on a non-distributed environment, but not for their inference capabilities.

The rest of the paper is organized as follows. The VIDECOM ontology is presented in Section 2. In Section 3 we present data storage mechanisms in KBR and the performance criteria for our comparative study. In Section 4 we present results of our experiments. We present some related work in Section 5. In Section 6 we conclude the paper and propose some future work.

2. THE VIDECOM ONTOLOGY

One important contribution of this paper is the VIDECOM ontology. VIDECOM is based on a deductive triple store composed of two parts. The first part is a domain ontology built on RDFS triple patterns extended with rules expressing domain knowledge. The second part is a populated ontology consisting of triples coming from trace events and the saturation mechanism.

Domain ontology. In this section, we briefly present some classes and properties of VIDECOM. We also present how developers can enrich VIDECOM using their knowledge about expected and unexpected behaviors.

Trace captures events that occurred during execution, such as interrupts, task running and context switches. Each event carries basic information like the start time, the duration, the task or the interrupt executed, the processor, the function called and arguments used. Table 1 shows an example of 8 trace events. The first event starts at timestamp *3771* and ends at timestamp *3781*. It corresponds to a *sys_read* operation executed by the task *ts_record* on *cpu* 0 with argument *0x46d*.

id	Start	End	Operation	Task	CPU	Arg
0	3771	3781	sys_read	ts_record	0	0x46d
1	3792	3873	sys_write	ts_record	0	0x11b
2	3879		switch_to	sshd	1	
3	3884	3885	Interrupt	mdtp_1	1	
4	4260		switch_to	kworker	1	0
5	4502	4602	sys_poll	ts_record	0	0xc1c
6	4605	4647	sys_read	ts_record	0	0x11d
7	4792	4873	sys_write	ts_record	0	0xe1e

Table 1: Illustration of 8 events from a real trace.

VIDECOM is based on a lightweight ontology of 608 classes and 238 properties. Figure 1 shows some of these classes and properties. The main class *Event* represents different types of events, such as *TASK_RUNNING* and *CONTEXT_SWITCH*. Properties *eventStartAt* and *eventEndAt* identify the start and the end timestamps of the event. The property *isExecutedOn* indicates the processor on which the event occurred, and the property *runningTask* indicates the task executed. The property *requestComponent* represents the *software component* (interrupt, system call or function call) requested by the event. The order between events is provided by *eventPrecedeInTrace* and *eventPrecedeInCPU* that indicate the order in trace and on each CPU. Moreover, *eventPrecedeOccurrence* indicates the order between

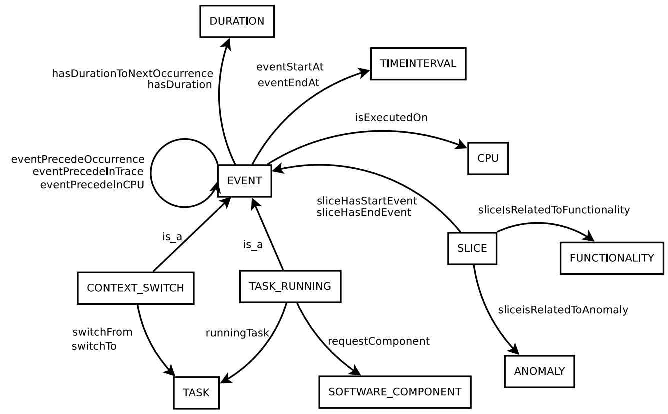


Figure 1: Some classes and properties of VIDECOM

occurrences of the same event. The property *hasDuration* indicates the duration of the event, and *hasDurationToNextOccurrence* represents the duration between consecutive occurrences of an event.

Some classes represent program behaviors concepts. The class $FUNCTIONALITY$ represents expected behaviors and $ANOMALY$ represents unexpected ones. To locate those behaviors, VIDECOM uses the class *SLICE* that represents a portion of a trace. A *SLICE* is bounded by two events using *sliceHasStartEvent* and *sliceHasEndEvent* properties. A *SLICE* can be related to a $FUNCTIONALITY$ (respectively an $ANOMALY$) with the property *sliceIsRelatedToFunctionality* (respectively *sliceIsRelatedToAnomaly*).

In VIDECOM, RDF triple patterns can be enriched with Datalog rules of the form $\phi \Rightarrow \psi$, where $\phi = \phi_1 \wedge ... \wedge \phi_n$ is a conjunction of atoms (triple patterns) called *premise*, and $\psi = \{\psi_1, ..., \psi_m\}$ is a conjunction of atoms called *conclusion*. An atom is a triple with variables at *subject*, *property* or *object* positions. The variable identifiers start with *?*. An atom holds if there are triples that matched the triple form of the atom. Therefore, the inference rule works as follows. If the conjunction of atoms of the *premise* holds, then the set of triples from the *conclusion* is added to the triple store. For example, let's consider the following inference rule

⟨?e,runningTask,?t⟩ ⇒ ⟨?e,rdf:type,TASK_RUNNING⟩,

if there is a triple in the triple store with *runningTask* in the *property* position and whatever in the *subject* (*?e*) and the *object* (*?t*) positions, then a triple is built and added to the triple store based on the atom of the *conclusion* with the corresponding value of variable *?e*.

We consider safe rules, that means that all variables in the *conclusion* are also in the *premise*, even the so-called blank node that is interpreted here as a constant for the safety property. RDFS semantics are captured by inference rules listed in the W3C recommendation[1]. Rules expressing domain knowledge corresponding to program behaviors are also captured using the same type of rules.

Populated ontology. Instances of VIDECOM classes and properties are built from events and stored as RDF triples ⟨*subject*, *property*, *object*⟩ in a triple store. Table 2 shows 10 RDF triples that represent the basic information contained in *event1* from table 1.

[1]http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/#rdfs-entailment

id	⟨ subject, property, object ⟩
1	⟨ *event1*, *eventStartAt*, "3792" ⟩
2	⟨ *event1*, *eventEndAt*, "3873" ⟩
3	⟨ *event1*, *isExecutedOn*, cpu0 ⟩
4	⟨ *event1*, *runningTask*, ts_record ⟩
5	⟨ *event1*, *requestComponent*, sys_write ⟩
6	⟨ *event1*, *eventPrecedeInTrace*, *event2* ⟩
7	⟨ *event1*, *eventPrecedeInCPU*, *event4* ⟩
8	⟨ *event1*, *eventPrecedeOccurrence*, *event7* ⟩
9	⟨ *event1*, *hasDuration*, "81" ⟩
10	⟨ *event1*, *hasDurationToNextOccurrence*, "1000" ⟩

Table 2: Set of RDF triples representing the basic information contained in *event1* from table 1.

The *saturation* ensures the completeness as well as the soundness of the query answering. It is done through an inference engine called *reasoner*. The reasoner implements a forward-chaining algorithm that applies all users and rdfs inference rules to populate the triple store with new facts. The saturated triple store is loaded in a KBR for querying purpose. In the next section we will briefly describe state-of-the-art KBRs.

3. KBR DESCRIPTION

In this section, we will briefly present several state-of-the-art KBRs. We classify the KBRs by the data storage mechanism they use to store the RDF triples.

3.1 Data storage mechanism

Various data storage layouts are presented in [6]. They distinguished *native* and *non-native* storage.

3.1.1 Native storage

This solution provides a way to store RDF triples in a model similar to the graph model. These solutions can be classified as *persistent disk-based* and *main memory-based*.

The *persistent disk-based* storage of RDF triples uses proprietary file format in many cases. Among the existing solutions we can mention *tdb*, *Sesame-native* and *rdf-3x*. *tdb*[2] uses a file system and stores triples in B+ Tree data structures. *Sesame-native* uses dedicated on-disk data structures for storage [14]. *rdf-3x* stores all the triples in a compressed clustered B+ Tree and uses an exhaustive index for all permutations of subject-property-object triples [12].

The *main memory-based* storage of RDF triples allocates a certain amount of the available main memory to store the whole RDF graph structure. *Jena* [10] and *Sesame-memory* [5] fall into this category.

3.1.2 Non native storage

The non-native storage solution makes use of Relational Database Management Systems (RDBMS) to store RDF triples. Storage of RDF triples in RDBMS exists in three models: *triple table*, *property table* and *vertical partitioning*.

In the *triple table* approach RDF triples are stored under the form $\langle subject, property, object\rangle$ in one large table with a three-columns schema corresponding to subject, property and object. Usual RDBMS indexes are built on each column to optimize access. *sdb*[3] is an example of this solution.

[2]http://jena.apache.org/documentation/tdb/architecture.html
[3]http://jena.apache.org/documentation/sdb/

The *property table* technique improves triples organization by allowing multiple triple patterns referencing the same subject to be retrieved with less join. In this model, triples are physically stored in a representation close to traditional relational schema in order to speed up the queries over the triple stores. In this approach each named table includes a subject and several fixed properties. The main idea is to discover clusters of subjects that appear frequently with the same set of properties. *4store* uses this approach [9].

Abadi et *al* suggested the *vertical partitioning* as an alternative to the property table. They illustrated the approach in *swStore* [1]. In this approach the triple table is divided into n two-columns tables, one table for each property in the data. In each of these tables, the first column contains the *subject* and the second column contains the *object* value related to this subject. Tables are stored by using a column-oriented RDBMS as a collection of columns rather than a collection of rows.

3.2 Inference support

Not all KBRs provide an RDFS *reasoner*. In those that we cited above only *Jena* and *Sesame* provide reasoners. The *Jena* reasoner implements a configurable subset of RDFS inference rules using the *RETE* algorithm [7] for forward chaining. We retained *Jena* reasoner because it is the only one that allows the implementation of user defined inference rules.

3.3 Performance criteria

We consider the following performances criteria to test scalability of KBR: the *saturation time* which is the time spent to saturate the triple store, the *loading time* which is the time spent to load the saturated triple store into the repository, and the *query response time* which is the time spent to answer a query.

We consider various characteristics of queries. We first consider the *selectivity*, because a high selective query must efficiently return a small portion of the entire triple store as answer. Next we consider the *k-complexity* defined as the number of atoms with k variables in the query. A *1-complexity* atom has one variable and a *2-complexity* atom has variables in two positions. We do not consider the case where a variable appears at the *property* position because it concerns very infrequent category of queries. Moreover a conjunction between two atoms with at least a variable in common in different places will indicate a *join*. We next consider whether or not the query uses *sorting* operators on the result. Indeed, due to the temporal nature of events, results may need to be sorted to facilitate their exploitation.

4. RESULTS AND DISCUSSIONS

In this section, we present a use case on STMicroelectronics MPSoC. This use case corresponds to an analysis of a real video recorder program called *ts_record*. We present experimental settings and results of our comparative study.

4.1 Experimental settings

We performed our experiments on 6 traces corresponding to different execution times of *ts_record*. Table 3 presents details on traces, such as, the execution duration, the number of runtime events recorded and the disk size of the trace. Section A of the Appendix provides more details on the *ts_record* use case, and Table 6 of the Appendix provides

Traces	Execution duration	# of events	# of triples
T0	2 m 38 sec	500 000	7 653 945
T1	3 m 25 sec	1 000 000	15 307 847
T2	7 m 43 sec	1 500 000	22 881 749
T3	9 m 23 sec	1 800 000	27 441 696
T4	10 m 25 sec	2 000 000	30 492 400
T5	25 m 47 sec	5 000 000	76 258 631

Table 3: List of traces and their corresponding execution time, number of runtime events and number of triples before saturation.

details on two users inference rules related to the *ts_record* use case that we used to enrich VIDECOM.

During the saturation, we ignored rdfs rules *rdfs1*, *rdfs4a*, *rdfs4b*, *rdfs12* and *rdfs13*, because they produced useless triples for trace analysis, such as *rdfs:Resource* and *rdfs:Literal*.

We consider 8 test queries from real analyst needs related to *ts_record* use case. Table 4 shows their characteristics. The query *Q0* is a conjunction of two atoms of *1-complexity* and *2-complexity*. More details on those queries are provided in the Appendix (Table 7).

	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
Selectivity	H	L	L	H	H	H	H	L
1-complexity	1	0	0	2	1	0	1	1
2-complexity	1	1	1	1	2	3	7	7
join	0	0	0	0	1	0	2	2
Filter	0	0	0	1	0	1	1	1
Order by			x					

Table 4: List of characteristics of our 8 test queries.

We performed our experiments on a machine with a 2.27 GHz Intel Xeon CPU and 64 GB of RAM. We chose the following non-commercial KBR for our comparative study: *jena* and *tdb*, *sesame-memory* and *sesame-native sdb*, and *vertical-mdb*. More details on their configuration are provided in the Appendix (Section B).

The reader can find the VIDECOM ontology, non-saturated and saturated public datasets[4], user inference rules, 8 tests queries and experimental results, on the website hosted at http://videcom.imag.fr.

4.2 Experimental results

4.2.1 The saturation time

Table 5 shows the saturation time for each trace, the number of triples after saturation and the disk size of the ontology stored in a N-Triple[5] format. The main memory was insufficient for *Jena* reasoner to saturate *T5*. We used a naive SQL-based implementation of forward-chaining algorithm in *vertical-mdb*. We saturated *T5* in 2 days 11 h 35 m 8 sec, but many optimizations are possible in our implementation to have better performances.

We observed that, one trace event produces *25* RDF triples, and that saturated triple store disk size is *68* times larger

Traces	Saturation time	# of triples	Size (MB)
T0	08 m 01 sec	13 051 370	2 185
T1	18 m 28 sec	25 981 602	4 347
T2	22 m 43 sec	38 790 013	6 508
T3	1 h 12 m 18 sec	46 494 600	7 818
T4	1 h 41 m 01 sec	51 607 411	8 670
T5	x	95 309 610	16 404

Table 5: Saturation time, number of triples and disk size of each trace after saturation.

than corresponding trace size. This result illustrates the difference of magnitude between the number of trace events and RDF triples, and the limitation of resource for saturation at this scale.

4.2.2 The loading time

Figure 2(a) shows the loading time for each repository and saturated triple store size. Indexes construction is included in the loading time. Because *vertical-mdb* uses batch import operations provided by MonetDB to copy data from file to tables and constructs index after data are loaded, it is the most efficient comparing to others condifered KBRs. It is 2 orders of magnitude faster than *sdb*, which builds indexes before loading data and, thus, frequently updates its indexes. Figure 2(a) also shows that *sesame-memory* and *sesame-native* cannot load 95 million triples. As it took *sdb* more than 2 hours to load 51 million triples (see Figure 2(a)), We were not able to load 95 million triples with *sdb*. In conclusion, results show that *vertical-mdb* can load 95 million triples in 2 minutes. We also observed that loading all the 95 million triples of *T5* with *Jena* filled all the main memory and an additional 6 GB space from the swap. Another fact is that being *non-persistent*, main-memory based repositories load data for each working session.

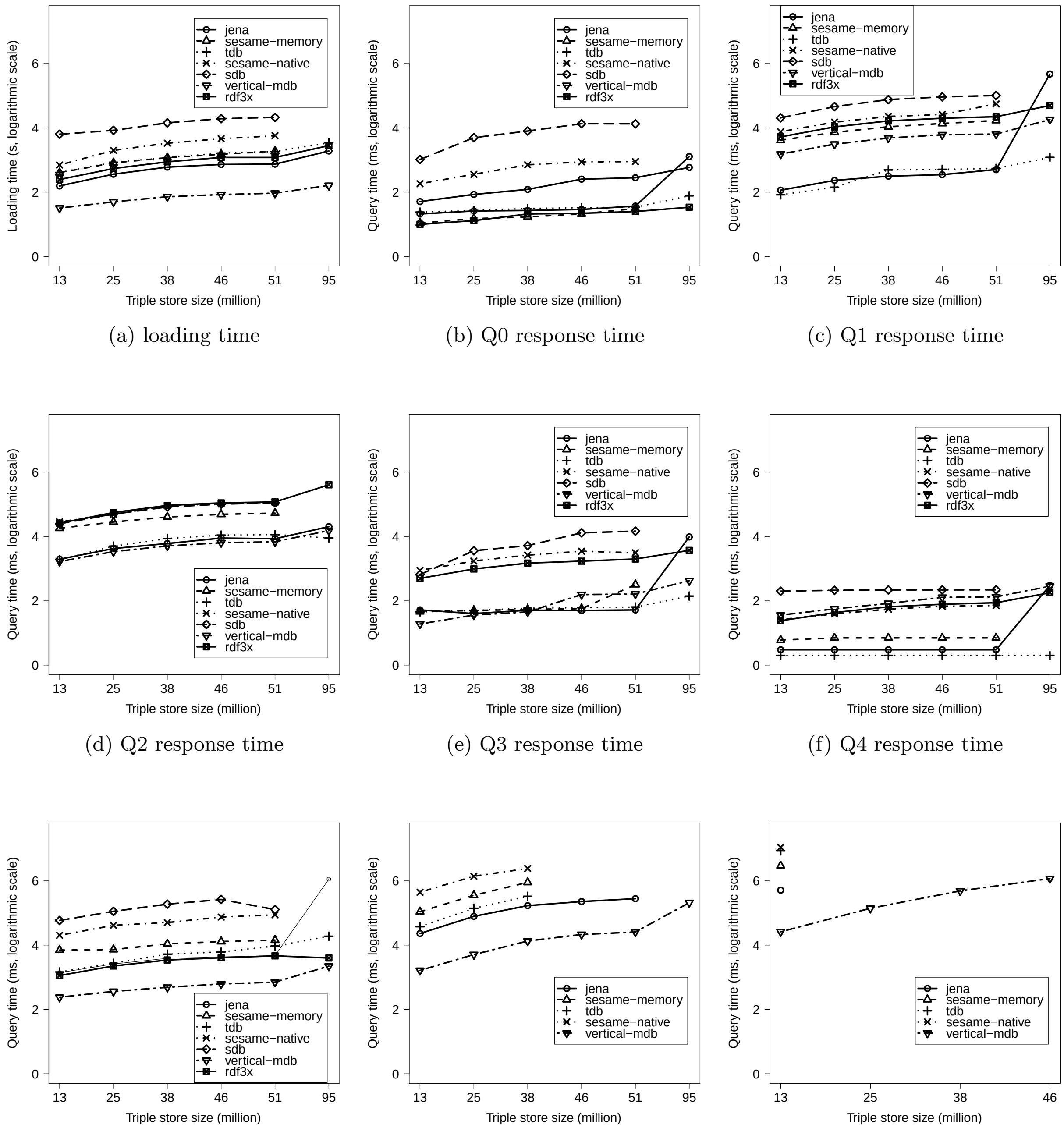
4.2.3 Query response time

We executed our queries on each KBR and for each trace from *T0* to *T5*. We ran each query 10 times and we collected the mean time as query response time.

Selectivity: Figure 2(b) shows response time for *Q0*. All KBR answered within 10 seconds. Thanks to their B+Tree based indexes *rdf-3x* and *tdb* are faster than the others. *Jena* scales poorly on 95 million triples because of *swap-in* and *swap-out* needed to get free space in main memory. In the case of *Q1* depicted in Figure 2(c), all KBR performance dropped by one order of magnitude but *rdf-3x* dropped by 2 orders of magnitude. However, *tdb* remains the fastest, which indicates that its indexes are well adapted for both high and low selectivity queries.

Sorting: Figure 2(d) shows the performance of *Q2*. We observed that performance of *vertical-mdb* did not change, unlike the others which lost at least one order of magnitude in response time. The reason can be the efficient implementation of sorting in *MonetDB*. We observed that the performance of *rdf-3x* dropped by 2 orders of magnitude, which indicates that the implementation of its sorting operators is not efficient. Figures 2(e) and 2(f) show the performance of *Q3* and *Q4*. Using inferred classes, *Q4* returns the same result as *Q3*. We observed that using inferred classes leads to fast query response time. The reason can be that inferred classes have higher selectivity.

[4]Because of the privacy of STMicrolectronics data, we provided datasets obtained from execution traces of video decoding on GStreamer (http://gstreamer.freedesktop.org/) an open source framework for multimedia applications
[5]http://www.w3.org/2001/sw/RDFCore/ntriples/



(a) loading time

(b) Q0 response time

(c) Q1 response time

(d) Q2 response time

(e) Q3 response time

(f) Q4 response time

(g) Q5 response time

(h) Q6 response time

(i) Q7 response time

Figure 2: Comparison of saturated triple store loading time and query response time for each KBR.

Interval of trace: Figure 2(g) shows the result when querying the same interval of time in all traces. We observed that *vertical-mdb* has better performance. The reason is that *vertical-mdb* identifies numerical *object* values; therefore, indexes built on columns containing numbers are more efficient than indexes built on strings like others do.

k-complexity: Figure 2(h) shows results for *Q5*. *rdf-3x* did not provide results due to an internal error in the query parser. Conjunctions are implemented in RDBMS as joins. In the case of *sdb* it consists of self-joins on the unique table Triple Table, and in the case of *vertical-mdb* it consists of joins between multiple tables. *sdb* failed to produce results; we suppose the reason being the inefficiency of self-join on large triple tables. *vertical-mdb* has acceptable response time (2 minutes for 95 million triples) unlike *tdb* which needed 5 minutes for 38 million triples. Figure 2(i) shows the performance of *Q7*. *vertical-mdb* has better response time (2 minutes for 46 million triples). Other KBRs performed poorly over 13 million triples. *Jena* took 8 minutes, *sesame-memory* took 48 minutes, *tdb* took 2 hours and *sesame-native* took 3 hours.

Discussion: Results show that the saturation with Jena is efficient but depends on the available memory. We also found that *tdb* indexes are efficient at large scale, and that *vertical-mdb* join implementation is efficient. Due to its fast loading speed *vertical-mdb* loads 95 million triples in 2 minutes and supports RDFS rule reasoning without memory limitation. The inefficiency in the saturation mechanism for *vertical-mdb* is mainly due to unoptimized code and far better performance should be expected. *vertical-mdb* is the only one that exhibited low running times accross all queries. It is also the only system that could handle Q7, a complex but realistic query. Considering the type of queries the analysts are interested in, the constraints of their practice, a vertical database system is the best solution in this context. Because an efficient inference engine is not required in the solution we chose based on a saturated triple dataset, some KBRs have been discarded. Nonetheless, KBRs such as, OWLIM and Virtuoso should be considered for future works considering distribution of the dataset and parallel processing.

5. RELATED WORK

Several RDF benchmarks were previously developed. We can cite, the Lehigh University Benchmark (LUBM) [8], the Berlin SPARQL Benchmark (BSBM)[4], and DBpedia [11]. Those benchmark handle large synthetic or real datasets (300 million of triples for DBpedia). They are mainly focused on the loading time and the query response time on various KBRs, such as, Jena, TDB, SDB, Sesame, virtuoso and OWLIM. Unlike those benchmarks, our benchmark is also focused on the saturation time of triples, because our datasets are deductive triple stores and need inference rule to be provide sound and complete query answers.

6. CONCLUSION AND FUTURE WORK

In this paper we presented a benchmark of triple stores scalability for MPSoC trace analysis. We presented VIDECOM, an ontology for trace analysis of application on MPSoC and we made a comparative study to test the scalability of 7 state-of-the-art KBRs. These experiments have shown that even with relatively simple traces and a large server, existing KBR have difficulties to scale up. Among the tested KBR, *vertical-mdb* is the only one that exhibited low running times across all queries. Given that execution traces are likely to grow much larger than the traces of these experiments, we can conclude that solutions based on a vertical approach will be required to handle them, and will have to be improved.

For our future work we are interested in giving answers to query over large traces in a fixed time. We plan to develop efficient approaches to speedup saturation, and we are interested in different strategies for query parallelization. Following this track we plan to consider OWLIM and Virtuoso KBRs for comparison with the vertical database approach.

7. ACKNOWLEDGEMENT

This work was supported by French FUI project SoC-Trace.

8. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: A Vertically Partitioned DBMS For Semantic Web Data Management. *The VLDB Journal*, 18(2):385–406, 2009.

[2] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, P. Senellart, et al. *Web Data Management*. Cambridge University Press, 2012.

[3] T. Ball. The Concept Of Dynamic Analysis. pages 216–234, 1999.

[4] C. Bizer and A. Schultz. Benchmarking the performance of storage systems that expose sparql endpoints. *World Wide Web Internet And Web Information Systems*, 2008.

[5] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, pages 197–203, 2003.

[6] C. David, C. Olivier, and B. Guillaume. A Survey Of RDF Storage Approaches. *ARIMA Journal*, 15:11–35, 2012.

[7] C. L. Forgy. Rete: A Fast Algorithm For The Many Pattern/many Object Pattern Match Problem. *Artificial intelligence*, 19(1):17–37, 1982.

[8] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark For OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[9] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design And Implementation Of A Clustered RDF Store. pages 94–109, 2009.

[10] B. McBride. Jena: Implementing The RDF Model and Syntax Specification. 2001.

[11] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Dbpedia sparql benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011*, pages 454–469. 2011.

[12] T. Neumann and G. Weikum. The RDF-3X Engine For Scalable Management of RDF Data. *The VLDB Journal*, 19(1):91–113, 2010.

[13] A. Roychoudhury. *Embedded Systems And Software Validation*. Morgan Kaufmann, 2009.

[14] Sesame. `http://www.openrdf.org/`, June 2014.

APPENDIX

A. THE TS_RECORD USE CASE

The program *ts_record* contains three tasks which can be scheduled on different CPUs of the MPSoC. The first task *t1* collects streaming data and stores them in small IP buffers. Every 100 milliseconds, the second task *t2* copies data from IP buffers to main memory. Finally, every 5 seconds, the last task *t3* copies them to a USB disk. The period between each task is important to avoid errors which can cause data loss. Based on this domain description, Table 6 presents 2 user inference rules that correspond to the behavior of task *t2*. For simplicity we present only two rules, but more rules can be added to VIDECOM. The number of user inference rules influence the saturation time. The rule *R1* instantiates the $FUNCTIONALITY$ subclass *sysWriteNormal* when two occurrences of events corresponding to *t2* are separated by a period equal to 100 milliseconds. The second rule *R2* instantiates the $ANOMALY$ subclass *sysWriteBlocked* if the period is greater than 100 milliseconds.

B. KNOWLEDGE BASE SYSTEMS FOR EXPERIMENTS

We chose the following non-commercial KBRs for our comparative study. *jena* and *tdb* (version 2.11.2) with their default configuration. We set the java heap size to 60 GB. We also chose *sesame-memory* and *sesame-native* (version 2.7.7), we configured *sesame-native* to support all the combination of subject-property-object indexes known as *spoc*, *posc*, and *opsc*. We set the java heap size to 60 GB. We chose *sdb* (version 1.3.4) backed on postgresql (version 9.3), and we configured *sdb* with the default "layout2" indexing storage. We set the java heap size to 60 GB.

Unfortunately *4Store* has not been maintained for 5 years and we were not able to install it in our setup configuration.

The column-store-based approach *swStore* implementation was not available. We implemented the approach as described in [1], but used MonetDB[6] (version 11.17.9) as a backend instead of C-Store because C-Store is no longer maintained. We called our implementation *vertical-mdb*.

[6]https://www.monetdb.org/

R1	⟨?e1, runningTask, ts_record⟩ ∧ ⟨?e1, requestComponent, sys_write⟩ ∧ ⟨?e1, eventPrecedeOccurrence, ?e2⟩ ∧ ⟨?e1, hasDurationToNextOccurrence, ?period⟩ ∧ (?period = 100)	⇒	⟨_:s, sliceHasStartEvent ?e1⟩, ⟨_:s, sliceHasEndEvent, ?e2⟩, ⟨_s:, sliceIsRelatedToFunctionality, sysWriteNormal⟩
R2	⟨?e1, runningTask, ts_record⟩ ∧ ⟨?e1, requestComponent, sys_write⟩ ∧ ⟨?e1, eventPrecedeOccurrence, ?e2⟩ ∧ ⟨?e1, hasDurationToNextOccurrence, ?period⟩ ∧ (?period > 100)	⇒	⟨_:s, sliceHasStartEvent ?e1⟩, ⟨_:s, sliceHasEndEvent, ?e2⟩, ⟨_s:, sliceIsRelatedToAnomaly, sysWriteBlocked⟩

Table 6: Two user inference rules to capture behavior of task *t2* in ts_record.

Queries	SPARQL
Q0	*finds all events which requested the program flush*

	`SELECT ?event ?debut`
	`WHERE {`
	`?event requestComponent flush_8_00 .`
	`?event eventStartAt ?debut .`
	`}`

Q1	*finds all events which corresponded to a context switch in the program*
	`SELECT ?event ?debut` `WHERE { ?event eventStartAt ?debut . }`
Q2	*finds all events which corresponded to a context switch in the program order by their start timestamp*
	`SELECT ?event ?debut` `WHERE { ?event eventStartAt ?debut . } ORDER BY ?event`
Q3	*finds all sys_write called by ts_record program which occurs more than 100 ms after the previous occurrence*

	`SELECT ?event ?duration`
	`WHERE {`
	`?event requestComponent sys_write .`
	`?event runningTask ts_record .`
	`?event hasDurationToNextOccurrence ?duration .`
	`FILTER (?duration > 100000)`
	`}`

Q4	*finds all events related to the concept sysWriteBlocked*

	`SELECT ?event ?duration`
	`WHERE {`
	`?slice sliceIsRelatedToAnomaly sysWriteBlocked .`
	`?slice sliceHasStartEvent ?event .`
	`?event hasDurationToNextOccurrence ?duration .`
	`}`

Q5	*finds all the tasks executed within timestamps 537756 and timestamp 19482669*

	`SELECT ?task`
	`WHERE {`
	`?event eventStartAt ?debut .`
	`?event eventEndAt ?end .`
	`?event runningTask ?task .`
	`FILTER (?debut >= 537756 AND ?end < 19482669)`
	`}`

Q6	*finds all tasks executed when a sysWriteBlocked occurred*

	`SELECT ?task`
	`WHERE {`
	`?slice sliceIsRelatedToAnomaly sysWriteBlocked .`
	`?slice sliceHasStartEvent ?event1 .  ?slice sliceHasEndEvent ?event2.`
	`?event1 eventStartAt ?sstart .  ?event2 eventEndAt ?send .`
	`?event eventStartAt ?debut .  ?event eventEndAt ?end .`
	`?event runningTask ?task .`
	`FILTER (?debut >= ?sstart AND ?end <= ?send)`
	`}`

Q7	*finds all tasks executed when a sysWriteNormal occurred*

	`SELECT ?task`
	`WHERE {`
	`?slice sliceIsRelatedToFunctionality sysWriteNormal .`
	`?slice sliceHasStartEvent ?event1 .  ?slice sliceHasEndEvent ?event2.`
	`?event1 eventStartAt ?sstart .  ?event2 eventEndAt ?send .`
	`?event eventStartAt ?debut .  ?event eventEndAt ?end .`
	`?event runningTask ?task .`
	`FILTER (?debut >= ?sstart AND ?end <= ?send)`
	`}`

Table 7: Test queries SPARQL description.