Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs

Jean Carlo Hamerski*†, Anderson R. P. Domingues*, Fernando G. Moraes*, Alexandre Amory*
*School of Technology – PUCRS – Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil
†Instituto Federal de Educação Ciência e Tecnologia do Rio Grande do Sul (IFRS), Porto Alegre, Brazil
jean.hamerski@restinga.ifrs.edu.br, anderson.domingues@acad.pucrs.br, {fernando.moraes,alexandre.amory}@pucrs.br

***Abstract*—Efficient serialization is a must-have feature in distributed embedded protocol stacks because of the restrained resources available for use in such systems. Although there are many serialization libraries out there, only some of them focus on resource usage, which is of most importance for the embedded domain. Thus, a comparison of serialization libraries considering resource usage is of high relevance for the embedded systems domain. This paper presents an experiment-based comparison of serialization libraries while concentrating on resource usage for a multiprocessor system-on-chip (MPSoC) platform. Results show that MsgPuck library surpasses other libraries for both serialization speed and memory consumption criteria.**

***Index Terms*—embedded systems, networking, serialization.**

I. Introduction and Background

A multiprocessor system-on-chip (MPSoC) typically consists of a set of processing elements (PEs) distributed over the chip and interconnected by a network-on-chip (NoC). In such systems, the memory organization is usually based on the No Remote Memory Access (NoRMA) approach, where the embedded processors indirectly access the remote address via messages sent via a NoC. As any other distributed system based on NoRMA, the communication infrastructure and protocol stack play an essential role in the system architecture.

Although the community widely debated on-chip physical communication infrastructure in the past [1], discussions about how to build a protocol stack for MPSoCs are rarer to be found. Regardless of the type of distributed domain, in typical protocol stacks, each of the layers has a set of protocols, which provide different services to the upper layer. The uppermost layer is the application layer while the lowermost one is the physical layer. Layers in between physical and application layers may vary from system to system. For instance, the OSI model suggests the implementation of five other layers: data link, network, transport, session, and presentation.

Despite the many different typical services for a protocol stack, serialization service is one of the most common among them. Two approaches for serialization are commonly found in the literature: binary and textual. In both, some serialization service is responsible for transforming some applications data structure into serial data, which can be either a stream of bytes (binary serialization) or merely a string (e.g., XML, JSON). Deserialization is the reverse process, where a stream of bytes (or a string) is received and converted to a copy of the data structure that originated it. It is important to note that serialization and deserialization are often called *serialization*, for short. We use this terminology for the rest of this paper.

There are dozens of serialization libraries available. However, these libraries are designed for a particular application domain that has requirements to be met. For example, applications for mobile phones require interoperability, low power consumption, among others. Another example is web applications where, usually, textual serialization formats (e.g., XML, YAML, and JSON) are preferred. Thus, the approach for serialization methods is bound to the domain of application.

In MPSoCs, where the amount of memory available for each PE is minimal (about tens or hundreds of KBytes), some programming platforms (e.g., Java, Python) cannot be used. It also excludes approaches that rely on large external dependencies (e.g., LibBoost). Besides, serialized data must be as small as possible to save bandwidth in the NoC, reducing network contention and energy consumption. For these reasons, string-based serialization (e.g., XML, JSON) are not valid options. Lastly, the serialization and deserialization must be fast, because the MPSoCs are usually based on small and simple processors. For instance, such processors work with 32-bit data path, fixed-point arithmetic, and three or five pipeline stages. Lastly, some applications have real-time constraints, although we do not address them in this paper.

The requirements for MPSoC excludes a substantial number of available serialization methods. The main contribution of this paper is to present a performance and resource usage evaluation of the few adequate solutions we could find, against the requirements mentioned before. As given in the next section, only a few studies present similar comparisons, but the solutions they evaluate have no use for MPSoCs since the domains of applications are radically different and less constrained when compared to the MPSoC domain.

II. Related Work

Although some studies on the comparison of serialization libraries exist in the literature, none of them are focused on serialization for highly memory constrained embedded platforms such as MPSoCs. The most related paper evaluates serialization with a focus on the Internet of Things domain [2]. However, they assume the Beagle Bone Black [3] or Odroid [4] hardware platforms. Both platforms have more than 512MB of memory and use a complete Linux-based OS, which has much more resources than the individual processors of our target MPSoC architecture. Thus, most of the evaluated serialization methods cannot be applied for MPSoCs because, for instance, the evaluated methods require from 1MB

978-1-5386-9562-3/18/$31.00 ©2018 IEEE

to 22MB of memory, which is more memory than the total amount of memory available for individual processors in the target MPSoC platform.

Maeda [5, 6] presented similar studies comparing several serialization libraries for Java. The requirement of languages such as Java and Python is also a limitation for processors with few KBytes of memory. Also, most of serializers work with formats such as XML, YAML, and JSON. Even though these formats present advantages concerning readability and interoperability, they present the drawback of bigger serialized data size when compared to the binary data formats. In the MPSoC domain, bandwidth usage is much more important than readability. For this reason, binary serialization is more suitable for MPSoC.

Sumaray and Makki [7] present a similar comparison for data size, serialization speed and ease of use of serialization libraries. However, they focus on Android-based platforms, which falls on the same issues as the previous references.

This paper is the first one to present an experiment-based comparison of serialization libraries applied to the context of embedded platform highly constrained in memory.

III. Serialization Libraries

In programming languages such as Java and C#, serialization is implemented by extending some interface of the built-in API. In other programming languages, such as C and C++, serialization must be implemented almost from scratch. The community developed several libraries, in the hope of mitigating efforts during the implementation of serialization.

Most of the libraries support both serializations of basic types and complex types. For the latter, the support mostly is provided through schemas, which the approach vary from library to library. Depending on the approach, resources required may exceed what is available on resource constrained platforms. We considered the following libraries.

MsgPack-c (version 2.1.5) is an implementation of the MsgPack (msgpack.org) serialization format with support to C and C++ language. MsgPack-c requires both a C++03 or C++11 compatible compiler and code annotation. MsgPack-c is available at github.com/msgpack/msgpack-c.

MsgPuck (v. 2.0) is a compact implementation of MsgPack library, written in C. MsgPuck repository announces interesting characteristics like zero-cost abstractions and zero overhead. All necessary library code is written in a pair of .c and .h files. In addition to support the base types, MsgPuck also has support to *arrays* as representation of a sequence of objects and *maps* for key-value pairs of objects. MsgPuck requires a C89+ or C++03 compatible compiler and it does not use schemas or code annotation. Available at github.com/rtsisyk/msgpuck.

MPack (v. 0.8.2) is a third implementation of MsgPack format, also written in C, without libc requirement. MPack does not use schemas or code annotation. Available at github.com/ludocode/mpack.

FlatBuffers (v. 1.8.0) allows that the data can be accessed without unpacking serialized data. However, this feature does not come without a cost, as show in the results of section VI. Flatbuffers is based on Protocol Buffer format (github.com/google/protobuf), also known as ProtoBuf. FlatBuffers requires both a C++11 compatible compiler and schemas definition. Available at github.com/google/flatbuffers.

NanoPB (v. 0.3.9) is an implementation of ProtoBuf that targets embedded systems. The serialization rely on schemas. Schemas of NanoPB are written into proto files, and their syntax is very similar to C's struct syntax. Nanopb should compile with most ansi-C compatible compilers, but it requires implementations of the strlen, memcpy and memset functions. Available at github.com/nanopb.

YAS (v. 5.0.1) is a replacement for Boost Serialization library (www.boost.org). Advantages of YAS include that it is header-only library and does not depends on external libraries and endianess. It requires both a C++11 compatible compiler and schema definitions. Available at github.com/niXman/yas.

IV. The Target Platform

A. Hardware Infrastructure

We perform the experiments in a 6x6 NoC-based MPSoC with homogeneous PEs. Each PE is equipped with an ARM's Cortex-M4F processor, direct memory access (DMA) module, a 512KBytes private random access memory (RAM), a network interface (NI), and router. An extra memory module is attached to the PE-0. This extra memory stores applications' static code until the system startup, when these are loaded into PEs' private memories. The platform is described using OVPSIM API (http://www.ovpworld.org/), which provides an instruction accurate simulation framework. Figure 1 presents the organization of the platform.

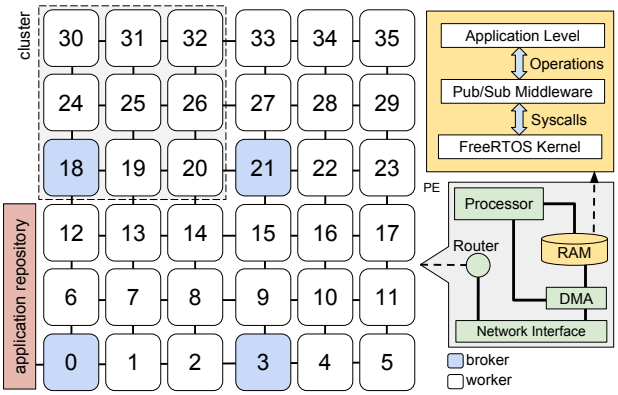

Figure 1. A 6x6 MPSoC platform with 3x3 clusters configuration.

B. Software and Network Infrastructure

The system running in each PE is composed of an extended version of FreeRTOS kernel [8] and a communication middleware [9] based on the publish-subscribe (pub-sub) pattern. The kernel is responsible for the task mapping and scheduling, and interaction with hardware. The network infrastructure counts with a protocol stack that interacts with the underlying NoC by system calls to the kernel. The pub-sub protocol is implemented by the middleware, intermediating communication

between publisher, subscriber, and broker nodes. The messages are transferred by using named data channels called *topics*.

V. Experimental Setup

A. Application Case Study

The experiments use a Producer-Consumer application, which contains two tasks: a producer and a consumer. The pub-sub middleware is used to perform the communication between the two tasks. A topic identifies the message flow. We use three distinct data types that represent the application data in each test scenario. The goal is to evaluate each serialization library using from straightforward to more complex data types. Figure 2 shows the used data types. The size of the structs A, B and C is, respectively, 8, 60 and 92 bytes, not considering the size of types that are dynamically sized. In our experiments, the size of all vectors is 1, that is, each vector has only 1 inserted element. The application data is delivered to the middleware level by using the provided API [9]. The middleware serializes the application data using a serialization library and transfers the serialized payload to the low level of the protocol stack until it is transmitted over the NoC. One message is sent by producer to consumer task in all scenarios.

```
/* Struct A */
struct Temperature{
  int32_t timestamp;
  float temp;};

/* Struct B */
struct InstrCnt{
  char[32] name;
  int32_t arith;
  int32_t logical;
  int32_t branch;
  int32_t jump;
  int32_t load;
  int32_t store;
  int32_t nop;};
```

```
enum TempLevel{High=1,Medium,Low};

/* Struct C */
struct AllSensors{
  std::string name;
  Temperature temp;
  float calib;
  int16_t cpu_usage;
  std::vector<uint8_t> occupancy;
  TempLevel  tempLevel;
  std::vector<InstrCnt> processors;
  InstrCnt instrCnt;
  std::vector<Temperature> History;};
```

Figure 2. Data structures used within the experiment.

B. Evaluated Metrics

Three metrics are extracted from each scenario: serialization and deserialization execution time, data size, and code size. A scenario is a combination of a serialization library and one of the three structures presented in Figure 2. Thus, a total of 16 scenarios were run, since YAS and NanoPB do not support the serialization of vectors, which are used in *Struct C*.

The **serialization execution time** corresponds to the total time spent by the producer process to perform the serialization of data. The **deserialization execution time** is the time spent by the consumer process to perform the deserialization of the received data. The measurement unit is the number of *clock cycles* measured through a timing model [10] that capture the executed instructions for each processor, generating an execution time from total executed instructions. The **data size** (DS) corresponds to the number of bytes required to encapsulate the serialized object into the packet payload, that is, the total size of the payload data of the application layer. The **code size** (CS) corresponds to the amount of memory required in the PE to store the software code. This metric considers the size of kernel, middleware and serialization library, together. The measurement unit is *bytes*. For comparison purposes, the size of the software code (kernel plus middleware) without any serialization library is 22.5KB.

VI. Results and Discussion

The performed experiment evaluates memory code size (CS) and payload data size (DS) achieved at each data struct type. Table I shows the results. Regarding CS and DS, the MsgPuck library has achieved the fittest result for all three data structures. All libraries present a larger code size for *Struct C* because this struct has elements with standard types to represent vectors and strings (*std::vector* and *std::string*). Consequently, the code size is significantly increased with additional methods to handle these types. An alternative to representing these object types would be the use of specialized libraries for embedded system, such as Embedded Template Library (http://www.etlcpp.com). FlatBuffers library needs a larger number of bytes to represent the serialized data, in addition to presenting the largest code size. FlatBuffers stores metadata of complex types into memory in a way that it serves as pointers to parts of the serialized data. In general, libraries that use schemas to represent data structures end up producing a larger code size. They use a run-time type identification (RTTI), which is a feature of the C++ programming language that exposes information about an object's data type at runtime. On the other hand, the libraries with smaller generated code size are those that require the explicit definition of the serialize/deserialize method for each object that composes the struct. We observe a trade-off between ease of use of the library and the amount of memory necessary to store the software. The system designer must keep this in mind when choosing the serialize library that fits into your design.

Table I
Memory size for each analyzed library.

Library	Struct A		Struct B		Struct C	
	CS	DS	CS	DS	CS	DS
MsgPack-c	168.7	9	269.1	15	272.6	67
MsgPuck	22.7	8	23	14	263.2	62
YAS	328.9	15	329.7	49	N/S[1]	N/S[1]
Flatbuffers	333.6	24	334	72	336	224
NanoPB	33.6	10	33.7	24	N/S[1]	N/S[1]
MPack	34.8	9	35	15	272.3	67

[1] N/S = No support for vector of structs

In order to demonstrate an example of ease of use, we show the code snippet necessary to serialize the *Struct B* in both the YAS library (Figure 4), that uses schemas, and the MsgPuck library (Figure 5), that requires the explicit declaration of serialization process for each struct element.

We observed that some serialization libraries (MsgPack-c, MsgPuck and MPack) consume more clock cycles for deserialization than serialization. In the Producer-Consumer application, specifically, this is very undesired, since, if the producer process is faster than the consumer, there will be a point in application's lifetime in which the consumer buffer will be full, and thus the producer will be unable to produce

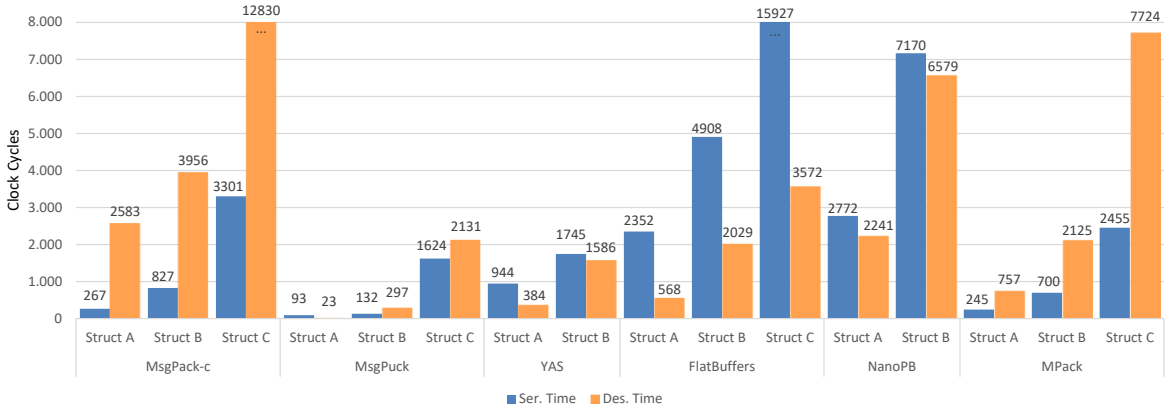

Figure 3. Serialization and deserialization execution time for each evaluated library and serialized data structs.

```
yas::mem_ostream os;
yas::binary_oarchive<yas::mem_ostream> oa(os);
oa & AppData;
this->message.msg_len = os.get_intrusive_buffer().size;
this->message.msg = (char*) os.get_intrusive_buffer().data;
-------------------------------------------------------
template<typename Ar>
void serialize(Ar &ar, const AppDataClass &t) {
ar & YAS_OBJECT_NVP( "InstCnt", ("n",t.InstrCnt.name),
("a",t.InstrCnt.arith), ("l",t.InstrCnt.logical),
"b",t.InstrCnt.branch), ("j",t.InstrCnt.jump),
("ld",t.InstrCnt.load), ("s",t.InstrCnt.store),
("n",t.InstrCnt.nop)); }
```

Figure 4. Code snippet of the YAS serialization process and required schema. AppData object contains an InstrCnt member corresponding to the Struct B.

```
char buf[MAX_PAYLOAD_SIZE];
char *w = buf;
w = mp_encode_str(w, AppData.InstrCnt.name,
strlen(AppData.InstrCnt.name));
w = mp_encode_int(w, AppData.InstrCnt.arith);
w = mp_encode_int(w, AppData.InstrCnt.logical);
w = mp_encode_int(w, AppData.InstrCnt.branch);
w = mp_encode_int(w, AppData.InstrCnt.jump);
w = mp_encode_int(w, AppData.InstrCnt.load);
w = mp_encode_int(w, AppData.InstrCnt.store);
w = mp_encode_int(w, AppData.InstrCnt.nop);
this->message.msg_len = strlen(buf);
this->message.msg = buf;
```

Figure 5. Code snippet of MsgPuck serialization process for the Struct B.

and deliver more packets. This behavior may slow the system, which is not tolerable in some domains (e.g. real-time applications). When the serialization time is greater than the deserialization time, the system is also slowed, but the node that hosts the consumer application will not be compromised in case it has other tasks to care of.

It is not the goal of this paper to analyze the cause of the high execution time values of the serialization/deserialization process in some library since it would require an internal analysis of each library's processes. We understand it to be of great value to disseminate the results presented in this paper as a benchmark, even for a small but important set of libraries.

VII. Conclusion and Future Work

In this paper we presented a comparison on serialization libraries while focusing on the MPSoC domain. Our results point that MsgPuck is the library that consumes less memory when serializing. Regarding serialization speed, MsgPuck also excels and beat other libraries. As future works we intend to evaluate serialization of data structures containing vector and string elements represented through specialized libraries for embedded system, such as Embedded Template Library. Also, this study can be extended to comprehend a more representative number of libraries by replicating the experiment.

Acknowledgment

The authors would like to thank Imperas Software and Open Virtual Platforms for their support and access to their models and simulator. Jean Carlo Hamerski is supported by CNPq and IFRS. Author Fernando Gehm Moraes is supported by FAPERGS (17/2551-0001196-1) and CNPq (302531/2016-5), Brazilian funding agencies. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior – Brasil (CAPES) - Finance Code 001.

References

[1] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, p. 1, 2006.

[2] B. Petersen, H. Bindner, S. You, and B. Poulsen, "Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things," in *2017 Computing Conference*, 2017, pp. 1339–1346.

[3] B. Foundation. BeagleBone Black. (Date last accessed 21-Sep-2018). [Online]. Available: https://beagleboard.org/black

[4] O. Platforms. Odroid. (Date last accessed 21-Sep-2018). [Online]. Available: https://www.hardkernel.com

[5] K. Maeda, "Comparative survey of object serialization techniques and the programming supports," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 5, no. 12, 2011.

[6] ——, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *DICTAP*, 2012, pp. 177–182.

[7] A. Sumaray and S. K. Makki, "A comparison of data serialization formats for optimal efficiency on a mobile platform," in *ICUIMC'12*. New York, NY, USA: ACM, 2012, pp. 48:1–48:6.

[8] G. Abich, M. G. Mandelli, F. R. Rosa, F. Moraes, L. Ost, and R. Reis, "Extending FreeRTOS to support dynamic and distributed mapping in multiprocessor systems," in *ICECS*, 2016, pp. 712–715.

[9] J. C. Hamerski, G. Abich, R. Reis, L. Ost, and A. Amory, "Publish-subscribe programming for a NoC-based multiprocessor system-on-chip," in *ISCAS*, 2017, pp. 1–4.

[10] F. Rosa, L. Ost, T. Raupp, F. Moraes, and R. Reis, "Fast energy evaluation of embedded applications for many-core systems," in *PATMOS*, 2014, pp. 1–6.