SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Fuel: A Fast General Purpose Object Graph Serializer

Martín Dias[3], Mariano Martinez Peck[1,2*], Stéphane Ducasse[1], Gabriela Arévalo[4,5]

[1] *RMoD Project-Team, Inria Lille–Nord Europe / Universite de Lille 1* [2] *Ecole des Mines de Douai* [3] *Universidad de Buenos Aires* [4] *Universidad Abierta Interamericana* [5] *CONICET*

SUMMARY

Since objects need to be stored and reloaded on different environments, serializing object graphs is a very important activity. There is a plethora of serialization frameworks with different requirements and design trade-offs. Most of them are based on recursive parsing of the object graphs, an approach which often is too slow. In addition, most of them prioritize a language-agnostic format instead of speed and language-specific object serialization. For the same reason, such serializers usually do not support features like class-shape changes, global references or executing pre and post load actions. Looking for speed, some frameworks are partially implemented at *Virtual Machine* (VM) level, hampering code portability and making them difficult to understand, maintain and extend.
In this paper we present Fuel, a general-purpose object serializer based on these principles: (1) speed, through a compact binary format and a pickling algorithm which invests time in serialization for obtaining the best performance on materialization; (2) good object-oriented design, without special help at VM; (3) serialize any object, thus have a full-featured language-specific format.
We implement and validate this approach in Pharo, where we demonstrate that Fuel is faster than other serializers, even those with special VM support. The extensibility of Fuel made possible to successfully serialize various objects: classes in Newspeak, debugger stacks, and full CMS object graphs. Copyright © 2011 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Object-Oriented Programming; Serialization; Object Graphs; Pickle Format; Smalltalk

1. INTRODUCTION

In object-oriented programming, since objects point to other objects, the runtime memory is an object graph. This graph of objects lives while the system is running and dies when the system is shutdown. However, sometimes it is necessary, for example, to backup a graph of objects into a non volatile memory so that it can be loaded back when necessary [14, 13, 5] or to export it so that the objects can be loaded in a different system. The same happens when doing migrations or when communicating with different systems. Besides this, databases normally need to serialize objects to write them to disk [8].

There are a lot of other possible uses for a serializer. For example, in case of remote objects, *e.g.,* remote method invocation and distributed systems [3, 9, 29], objects need to be serialized and passed around the network. A Version Control System that deals with code represented as first-class objects needs to serialize and materialize those objects: Parcels [17] is a typical example. Today's web applications need to store state in the HTTP sessions and move information between the client and the server.

*Correspondence to: marianopeck@gmail.com

Approaches and tools to export object graphs must scale to large object graphs as well as be efficient. However, most of the existing solutions do not solve this last issue properly. This is usually because there is a trade-off between speed and other quality attributes such as readability/independence from the encoding. For example, exporting to XML [25] or JSON [12] is more readable than exporting to a binary format since it can opened it and edited with any text editor. However, a good binary format is faster than a text based serializer when reading and writing. Some serializers like *pickle* [20] in Python or *Google Protocol Buffers* [21] let the user choose between text and binary representation. From our point of view, the following five main points shape the space of serializers for class-based object-oriented programming languages. We think all of them are important, yet not necessary in all scenarios. For example, one user can consider that speed is a critical requirement while portability not. Another user can require portability but not an outstanding performance.

1. Serializer *speed* is an important aspect since it enables more extreme scenarios such as saving objects to disk and loading them only on demand at the exact moment of their execution [13, 5].

2. Serializer *portability and customization*. Since many approaches are often too slow, Breg and Polychronopoulos advocate that object serialization should be done at the virtual machine level [7]. However, this implies non portability of the virtual machine and difficult maintenance. In addition, moving behavior to the VM level usually means that the serializer is not easy to customize or extend.

3. Another usual problem are *class* changes. For example, the class of a saved object can be changed after the object is saved. At writing time, the serializer should store all the necessary information related to class shape to deal with these changes. At loading time, objects must be updated in case it is required. Many object serializers are limited regarding this aspect. For example, the Java Serializer [11] does not support the modification of an object's hierarchy nor the removing of the implementation of the Serializable interface.

4. *Storing and loading policies*. Ungar [28] claims that the most important and complicated problem is not to detect the subgraph to export, but to detect the implicit information of the subgraph that is necessary to correctly load back the exported subgraph in another system. Examples of such information are (1) whether to export an actual value or a counterfactual initial value or (2) whether to create a new object in the new system or to refer to an existing one. In addition, it may be necessary that certain objects run some specific code once they are loaded in a new system.

5. *Completeness*. Serializers are often limited to certain kind of objects they save. For example, most of the Smalltalk serializers do not support serialization of objects like BlockClosure, MethodContext, CompiledMethod, etc. Now, in dynamic programming languages *e.g.,* Smalltalk, methods and classes are first class objects, *i.e.,* user's code is represented by objects. Similarly, the execution stack and closures are objects. The natural question is if we can use serializers as a code management system underlying mechanism. VisualWorks Smalltalk introduced a pickle format to save and load code called Parcels [17]. However, such infrastructure is more suitable for managing code than a general purpose object graph serializer.

This paper presents Fuel, a fast open-source general-purpose framework to serialize and deserialize object graphs using a pickle format which clusters similar objects.

Traditional serializers encode the objects of the graph while traversing it. The stream is a sequence of bytes where they store each object plus an identifier of its type. The unpickling then starts to read objects from the stream. For each object it reads, it needs to read its type as well as determine and interpret how to materialize that encoded object. In other words, the materialization is done *recursively*.

DOI: 10.1002/spe

In the contrary, in Fuel there is a first traversal of the graph (we call this phase "analysis") where each object is associated with an specific type which is called "cluster" in Fuel. Fuel first writes the instances and then the references. During materialization, Fuel first materializes the instances. Since all the objects of a cluster have the same type, Fuel reads that information in the stream only once. The materialization can be done in a bulk way which means that we can just iterate and instantiate the objects. Finally, Fuel iterates and sets the references for each of the materialized object. Fuel's materialization is done *iteratively*.

We show in detailed benchmarks that we have the best performance in most of the scenarios: For example, with a large binary tree as sample, Fuel is four times faster writing and seven times faster loading than its competitor SmartRefStream. If a slow stream is used *e.g.,* a file stream, Fuel is sixteen times faster for writing thanks to its internal buffering. We have implemented and validated this approach in Pharo [4] and Fuel has already been ported to other Smalltalk implementations. Fuel is also used to serialize classes in Newspeak. The pickle format presented in this paper is similar in spirit to the one of Parcels [17]. However, Fuel is not focused on code loading and is highly customizable to cope with different objects.

In addition, this article demonstrates the speed improvements made in comparison to traditional approaches. We demonstrate that we can build a fast serializer without specific VM support, with a clean object-oriented design and providing the most possible required features for a serializer.

The main contributions of the paper are: (1) Description of our pickle format and algorithm, (2) Description of the key implementation points, (3) Evaluation of the speed characteristics, and (4) Precise comparison of speed improvements with other serializers.

To avoid confusion, we define terms used in this paper. *Serializing* is the process of converting the whole object graph into a sequence of bytes. We consider the words *pickling* and *marshalling* as synonyms. *Materializing* is the inverse process of serializing, *i.e.,* regenerate the object graph from a sequence of byes. We consider the words *deserialize*, *unmarshalling* and *unpickling* as synonyms. We understand the same for *object serialization*, *object graph serialization* and *object subgraph serialization*. An object can be seen as a subgraph because of its pointers to other objects. At the same time, everything is a subgraph if we consider the whole memory as a large graph.

The remainder of the paper is structured as follows: In Section 2, we present our solution and an example of a simple serialization which illustrates the pickling format. We expose some elements to evaluate a serializer in Section 3 and we apply them to Fuel in Section 4. Section 5 gives an overview of Fuel's design. A large amount of benchmarks are provided in Section 6. We present Fuel real life usages in Section 7. Finally, we discuss related work in Section 8 and we conclude.

2. FUEL'S FOUNDATION

In this section we explain the most important characteristics of Fuel implementation that make a difference with traditional serializers.

2.1. Pickle format

Pickle formats are efficient formats to support *transport*, *marshalling* or *serialization* of objects [23]. Riggs defines: "*Pickling* is the process of creating a serialized representation of objects. Pickling defines the serialized form to include meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated. *Unpickling* is the complementary process of recreating objects from the serialized representation." (extracted from [23])

Fuel's pickle format works this way: during serialization, it first performs an analysis phase, which is a first traversal of the graph. During such traversal, each object is associated to a specific *cluster*. Then Fuel first writes the instances (vertexes in the object graph) and after that, the references (edges). While materializing, Fuel first materializes the instances. Since all the objects of a cluster have the same type, Fuel stores and reads that information in the stream only once. The

DOI: 10.1002/spe

materialization can be done in a bulk way (it can just iterate and instantiate the objects). Finally, Fuel iterates and set the references for each of the materialized objects.

Even if the main goal of Fuel is materialization speed, the benchmarks of Section 6 show we also have almost the best speed on serialization too.

Serializing a rectangle. To present the pickling format and algorithm in an intuitive way, we show below an example of how Fuel stores a rectangle.

In the following snippet, we create a rectangle with two points that define the origin and the corner. A rectangle is created and then passed to the serializer as an argument. In this case, the rectangle is the *root* of the graph which also includes the points that the rectangle references. The first step analyzes the graph starting from the root. Objects are mapped to *clusters* following some criteria. For this example we only use the criterium *by class*. In reality Fuel defines a set of other clusters such as *global objects* (it is at Smalltalk dictionary) or small integer range (*i.e.,* an integer is between 0 and $2^{32} - 1$) or *key literals* (nil true or false), etc.

```
| aRectangle anOrigin aCorner |
anOrigin := Point x: 10 y: 20.
aCorner := Point x: 30 y: 40.
aRectangle := Rectangle origin: anOrigin corner: aCorner.
(FLSerializer on: aFileStream) serialize: aRectangle.
```

Figure 1 illustrates how the rectangle is stored in the stream. The graph is encoded in four main sections: header, vertexes, edges and trailer. The Vertexes section collects the instances of the graph. The Edges section contains indexes to recreate the references between the instances. The trailer encodes the root: a reference to the rectangle.

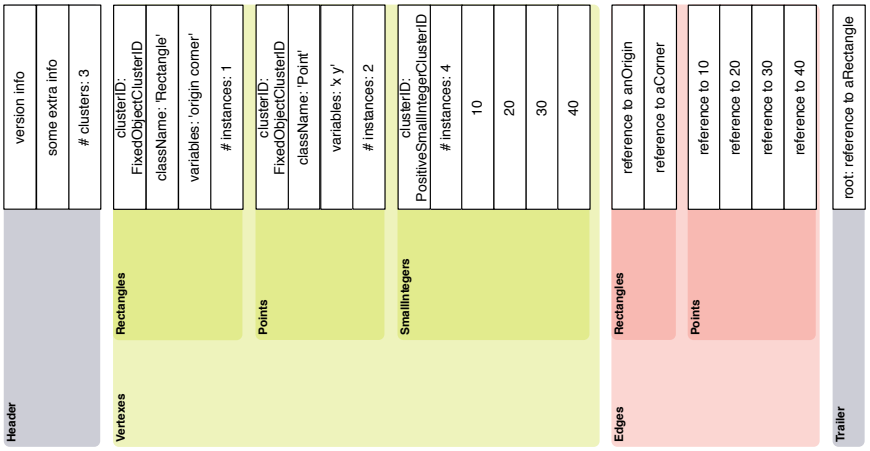

Figure 1. A graph example encoded with the pickle format.

At load time, the serializer processes all the clusters: it creates instances of rectangles, points, small integers in a batch way and then set the references between the created objects.

2.2. *Grouping objects in clusters*

Typically, serializers do not group objects. Thus, each object has to encode its type at serialization and decode it at deserialization. This is an overhead in time and space. In addition, to recreate each instance the serializer may need to fetch the instance's class.

The purpose of grouping similar objects is not only to reduce the overhead on the byte representation that is necessary to encode the *type* of the objects, but more importantly because the materialization can be done *iteratively*. The idea is that the type is encoded and decoded only once for all the objects of that type. Moreover, if recreation is needed, the operations can be grouped.

Prepared using speauth.cls
DOI: 10.1002/spe

The type of an object is sometimes directly mapped to its class but the relation is not always one to one. For example, if the object being serialized is Transcript, the type that will be assigned is the one that represents global objects. For speed reasons, we distinguish between positive SmallInteger and negative one. From Fuel's perspective, they are from different types.

In Fuel we have a class hierarchy of Clusters. Each one know how to encode and decode the objects they group. Here are some examples: PositiveSmallIntegerCluster groups positive instances of SmallInteger; NegativeSmallIntegerCluster groups negative instances of SmallInteger; FloatCluster groups Float instances. FixedObjectCluster is the cluster for regular classes with indexable instance variables that do not require any special serialization or materialization. One instance of this cluster is created for each class.

In Figure 1, there is one instance of PositiveSmallIntegerCluster and two instances of FixedObjectCluster, one for each class (Rectangle and Point). Such clusters will contain all the respective instances of the classes they represent.

Clusters decide not only *what* is encoded and decoded but also *how*. For example, FixedObjectCluster writes into the stream a reference to the class whose instances it groups and, then, it writes the instance variable names. In contrast, FloatCluster, PositiveSmallIntegerCluster or NegativeSmallIntegerCluster do not store such information because it is implicit in the cluster implementation.

In Figure 1, for small integers, the cluster directly writes the numbers 10, 20, 30 and 40 in the *Vertexes* part of the stream. However, the clusters for Rectangle and Point do not write the objects in the stream. This is because such objects are no more than just references to other objects. Hence, only their references are written in the *Edges* part. In contrast, there are objects that contain self contained state, *i.e.,* objects that do not have references to other objects. Examples are Float, SmallInteger, String, ByteArray, LargePositiveInteger, etc. In those cases, the cluster associated to them have to write those values in the *Vertexes* part of the stream.

The way to specify custom serialization or materialization of objects is by creating specific clusters.

2.3. Analysis phase

The common approach to serialize a graph is to traverse it and, while doing so, to encode the objects into a stream. Since Fuel groups similar objects in clusters, it needs to traverse the graph and associate each object to its correct cluster. As explained, that fact significantly improves the materialization performance. Hence, Fuel does not have one single phase of traverse and writing, but instead two phases: analysis and writing. The analysis phase has several responsibilities:

- It takes care of traversing the object graph and it associates each object to its cluster. Each cluster has a corresponding list of objects which are added there while they are analyzed.
- It checks whether an object has already been analyzed or not. Fuel supports cycles (an object is only written once even if it is referenced from several objects in the graph).
- It gives support for global objects, *i.e.,* objects which are considered global are not written into the stream. Instead the serializer stores the minimal needed information to get the reference back at materialization time. Consider as an example the objects that are in Smalltalk globals. If there are objects in the graph referencing *e.g.,* the instance Transcript, we do not want to serialize that instance. Instead, we just store its global name to get the reference back during materialization. The same happens with the Smalltalk class pools.

Once the analysis phase is over, the writing follows: it iterates over the clusters and, for each it writes its objects.

2.4. Two phases for writing instances and references.

The encoding of objects is divided in two parts: (1) instances writing and (2) references writing. The first phase includes just the minimal information needed to recreate the instances *i.e.,* the vertexes of the graph. The second phase has the information to recreate references that connect the instances *i.e.,* the edges of the graph.

DOI: 10.1002/spe

Notice that these two phases are mandatory to be able to perform the bulk materialization. If this division does not exist, the serializer cannot do a bulk materialization because to materialize an object it needs to materialize its instance variables, which of course can be of a different type.

In the materialization, there are two phases as well, the first one for materializing the instances and the second one to set the references between the materialized objects.

2.5. Iterative graph recreation

This is the most important characteristic of Fuel's pickle format. Other characteristics such as the analysis phase, grouping instances in clusters, and having two phases for serializing/materializing instances and references, are all necessary to achieve iterative graph recreation.

During Fuel serialization, when a cluster is serialized, the amount of objects of such cluster is stored as well as the total amount of objects of the whole graph. This means that, at materialization time, Fuel knows exactly the number of allocations (new objects) needed for each cluster. For example, one Fuel file can contain 17 large integers, 5 floats, 5 symbols, etc. In addition, for variable objects, Fuel also stores the size of such objects. So, for example, it does not only know that there are 5 symbols but also that the first symbol is size 4, the second one is 20, the third is 6, etc.

Therefore, the materialization populates an object table with indices from 1 to N where N is the number of objects in the file. Most serializers determine which object to create as they walk a (flattened) input graph. In the case of Fuel, it does so in batch (spinning in a loop creating N instances of each class in turn).

Once that is done, the objects have been materialized but updating the references is pending, *i.e.,* which fields refer to which objects. Again, the materializer can spin filling in fields from the reference data instead of determining whether to instantiate an object or dereference an object ID as it walks the input graph.

This is the main reason why materializing is much faster in Fuel than in other approaches.

2.6. Breadth-first traversal

Most of the serializers use a depth-first traversal mechanism to serialize the object graph. Such mechanism consists of a simple recursion:

1. Take an object and look it up in a table.

2. If the object is in the table, it means that it has already been serialized. Then, we take a reference from the table and write it down. If it is not present in the table, it means that the object has not been serialized and that its contents needs to be written. After that, the object is serialized and a reference representation is written into the table.

3. While writing the contents, *e.g.,* instance variables of an object, the serializer can encounter simple objects such as instances of String, SmallInteger, LargePositiveInteger, ByteArray or complex objects (objects which have instance variables that refer to other objects). In the latter case, we start over from the first step.

This mechanism can consume too much memory depending on the graph, *e.g.,* its depth, the memory to hold all the call stack of the recursion can be too much.

In Fuel, we do not use the mentioned depth first recursion but instead we do a breadth-first traversal. The difference is mainly in the last step of the algorithm. When an object has references to other objects, instead of following the recursion to analyze these objects, we just push such objects on a custom stack. Then, we pop objects from the stack and analyze them. The routine is to pop and analyze elements until the stack is empty. In addition, to improve even more speed, Fuel has its own SimpleStack class implementation. With this approach, the resulting stack size is much smaller and the memory footprint is smaller as well. At the same time, we decrease serialization time by 10%.

This is possible because Fuel has a two phases for writing instances and references as explained above.

DOI: 10.1002/spe

3. SERIALIZER REQUIRED CONCERNS AND CHALLENGES

Before presenting Fuel's features in more detail, we present some useful elements of comparison between serializers. This list is not exhaustive.

3.1. Serializer concerns

Below we list general aspects to analyze in a serializer.

Performance. In almost every software component, time and space efficiency is a wish or sometimes even a requirement. It does become a need when the serialization or materialization is frequent or when working with large graphs. We can measure both speed and memory usage, either serializing and materializing, as well as the size of the obtained stream. We should also take into account the initialization time, which is important when doing frequent small serializations.

Completeness. It refers to what kind of objects the serializer can handle. It is clear that it does not make sense to transport instances of some classes, like FileStream or Socket. Nevertheless, serializers often have limitations that restrict use cases. For example, an apparently simple object like a SortedCollection usually represents a challenging graph to store: it references a block closure which refers to a method context and most serializers do not support transporting them, often due to portability reasons. In view of this difficulty, it is common that serializers simplify collections storing them just as a list of elements.

In addition, in comparison with other popular environments, the object graphs that one can serialize in Smalltalk are much more complex because of the reification of metalevel elements such as methods, block closures, and even the execution stack. Usual serializers are specialized for plain objects or metalevel entities (usually when their goal is code management), but not both at the same time.

Portability. Two aspects related to portability. One is related to the ability to use the same serializer in different dialects of the same language or even a different language. The second aspect is related to the ability of being able to materialize in a dialect or language a stream which was serialized in another language. This aspect brings even more problems and challenges to the first one.

As every language and environment has its own particularities, there is a trade-off between portability and completeness. Float and BlockClosure instances often have incompatibility problems.

For example, Action Message Format [1], Google Protocol Buffers [21], Oracle Coherence*Web [19], Hessian [10], have low-level language-independent formats oriented to exchange structured data between many languages. In contrast, SmartRefStream in Pharo and Pickle [20] in Python choose to be language-dependent but enabling serialization of more complex object graphs.

Security. Materializing from an untrusted stream is a possible security problem. When loading a graph, some kind of dangerous objects can enter to the environment. The user may want to control in some way what is being materialized.

Atomicity. We have this concern expressed in two parts: for saving and for loading. As we know, the environment is full of mutable objects *i.e.,* that change their state over the time. So, while the serialization process is running, it is desired that such mutable graph is written in an atomic snapshot and not a potential inconsistent one. On the other hand, if we load from a broken stream, it will not successfully complete the process. In such case, no secondary effects should affect the environment. For example, there can be an error in the middle of the materialization which means that certain objects have already been materialized.

Versatility. Let us assume a class is referenced from the graph to serialize. Sometimes we may be interested in storing just the name of the class because we know it will be present when materializing the graph. However, sometimes we want to really store the class with full detail, including its method

Prepared using speauth.cls
DOI: 10.1002/spe

dictionary, methods, class variables, etc. When serializing a package, we are interested in a mixture of both: for external classes, just the name but, for the internal ones, full detail.

This means that given an object graph, there is not an unique way of serializing it. A serializer may offer the user dynamic or static mechanisms to customize this behavior.

3.2. Serializer challenges

The following is a list of concrete issues and features that users can require from a serializer.

Cyclic object graphs and duplicates. Since the object graph to serialize usually has cycles, it is important to detect them and to preserve the objects' identity. Supporting this means decreasing the performance and increasing the memory usage, because for each object in the graph it is necessary to check whether it has been already processed or not, and if it has not, it must be temporally stored.

Maintaining identity. There are objects in the environment we do not want to replicate on deserialization because they represent well-known instances.

We can illustrate with the example of Transcript, which in Pharo environment is a global variable that binds to an instance of ThreadSafeStream. Since every enviroment has its own unique-instance of Transcript, the materialization of it should respect this characteristic and thus not create another instance of ThreadSafeStream but use the already present one.

Transient values. Sometimes, objects have a temporal state that we do not want to store and we want also an initial value when loading. A typical case is serializing an object that has an instance variable with a lazy-initialized value. Suppose we prefer not to store the actual value. In this sense, declaring a variable as *transient* is a way of delimiting the graph to serialize.

There are different levels of transient values:

- Instance level: When only one particular object is transient. All objects in the graph that are referencing to such object will be serialized with a nil in their instance variable that points to the transient object.
- Class level: Imagine we can define that a certain class is transient in which case all its instances are considered transient.
- Instance variable names: the user can define that certain instance variables of a class have to be transient. This means that all instances of such class will consider those instance variables as transient. This type of transient value is the most common.
- List of objects: the ability to consider an object to be transient only if it is found in a specific list of objects. The user should be able to add and remove elements from that list.

Class shape change tolerance. Often, we need to load instances of a class in an environment where its definition has changed. The expected behavior may be to adapt the old-shaped instances automatically when possible. We can see some examples of this in Figure 2. For instance variable position change, the adaptation is straightforward. For example, version v2 of Point changes the order between the instance variables x and y. For the variable addition, an easy solution is to fill with nil. Version v3 adds instance variable distanceToZero. If the serializer also lets one write custom messages to be sent by the serializer once the materialization is finished, the user can benefit from this hook to initialize the new instance variables to something different that nil.

In contrast to the previous examples, for variable renaming, the user must specify what to do. This can be done via hook methods or, more dynamically, via materialization settings.

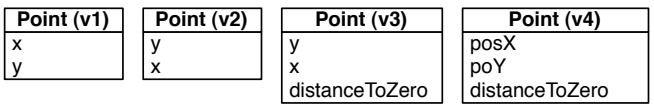

Figure 2. Several kinds of class shape changing.

DOI: 10.1002/spe

There are even more kinds of changes such as adding, removing or renaming a class or an instance variable, changing the superclass, etc. As far as we know, no serializer fully manages all these kinds of changes. Actually, most of them have a limited number of supported change types. For example, the Java Serializer [11] does not support changing an object's hierarchy or removing the implementation of the Serializable interface.

Custom reading. When working with large graphs or when there is a large number of stored streams, it makes sense to read the serialized bytes in customized ways, not necessarily materializing all the objects as we usually do. For example, if there are methods written in the streams, we may want to look for references to certain message selectors. Maybe we want to count how many instances of certain class we have stored. We may also want to list the classes or packages referenced from a stream or even extract any kind of statistics about the stored objects.

Partial loading. In some scenarios, especially when working with large graphs, it may be necessary to materialize only a part of the graph from the stream instead of the whole graph. Therefore, it is a good feature to simply get a subgraph with some holes filled with nil or even with proxy objects to support some kind of lazy loading.

Versioning. The user may need to load an object graph stored with a different version of the serializer. This feature enables version checking so that future versions can detect that a stream was stored using another version and act consequently: when possible, migrating it and, when not, throwing an error message. This feature brings the point of backward compatibility and migration between versions.

4. FUEL'S FEATURES

In this section, we analyze Fuel in accordance with the concerns and features defined in Section 3.

4.1. Fuel serializer concerns

Performance. We achieved an excellent time performance. The main reason behind Fuel's performance in materialization is the ability to perform the materialization *iteratively* rather than *recursively*. That is possible thanks to the clustered pickle format. Nevertheless, there are more reasons behind Fuel's performance:

- We have implemented special collections to take advantage of the characteristics of algorithms.
- Since Fuel algorithms are iterative, we know *in advance* the size of each loop. Hence, we can always use optimized methods like to:do: for the loops.
- For each basic type of object such as Bitmap, ByteString, ByteSymbol, Character, Date, DateAndTime, Duration, Float, Time, etc. , we optimize the way they are encoded and decoded.
- Fuel takes benefits of being platform-specific (Pharo), while other serializers sacrifice speed in pursuit of portability.

Performance is extensively studied and compared in Section 6.

Completeness. We are close to say that Fuel deals with all kinds of objects available in a Smalltalk runtime. Note the difference between being able to serialize and getting something meaningful while materializing. For example, Fuel can serialize and materialize instances of Socket, Process or FileStream but it is not sure they will still be valid once they are materialized. For example, the operating system may have given the socket address to another process, the file associated to the file stream may have been removed, etc. There is no magic. Fuel provides hooks to solve the mentioned problems. For example, there is a hook so that a message is send once the materialization is done. One can implement the necessary behavior to get a meaningful object. For instance, a new

Prepared using speauth.cls
DOI: 10.1002/spe

socket may be assigned. Nevertheless sometimes there is nothing to do, *e.g.,* if the file of the file stream was removed by the operating system. Note that some well known special objects are treated as external references because that is the expected behavior for a serializer. Some examples are, Smalltalk, Transcript and Processor.

Portability. As we explained in other sections, the portability of Fuel's source code is not our main focus. However, Fuel has already been successfully ported to Squeak, to Newspeak programming language and, at the moment of this writing, half ported to VisualWorks. What Fuel does not support is the ability to materialize in a dialect or language a stream which was serialized in another language. We do not plan do to communicate with another language.

Even if Fuel's code is not portable to other programming languages, the algorithms and principles are general enough for being reused in other object environments. In fact, we have not invented this type of pickle format that groups similar objects together and that does a iterative materialization. There are already several serializers that are based on this principle such as Parcels serializer from VisualWorks Smalltalk.

Versatility. Our default behavior is to reproduce the serialized object as exact as possible. Nonetheless, for customizing that behavior we provide what we call *substitutions* in the graph. The user has two alternative to do it: at class-level or at instance-level. In the former case, the class implements hook methods that specify that its instances will be serialized as another object. In the latter, the user can tell the serializer that when an object (independently of its class) satisfies certain condition, then it will be serialized as another object.

Security. Our goal is to give the user the possibility to configure validation rules to be applied over the graph (ideally) before having any secondary effect on the environment. This has not been implemented yet.

Atomicity. Fuel can have problems if the graph changes during the analysis or serialization phase. Not only Fuel suffers this problem, but also the rest of the serializers we analyzed. From our point of view, the solution always lies at a higher level than the serializer. For example, if one has a domain model that is changing and wants to implement save/restore, one needs to provide synchronization so that the snapshots are taken at valid times and that the restore actions work correctly.

4.2. Fuel serializer challenges

In this section, we explain how Fuel implements some of the features previously commented. There are some mentioned challenges such as *partial loading* or *custom reading* that we do not include here because Fuel does not support them at the moment.

Cyclic object graphs and duplicates. Fuel checks that every object of the graph is visited once supporting both cycle and duplicate detection.

Maintaining identity. The default behavior when traversing a graph is to recognize some objects as *external references*: Classes registered in Smalltalk, global objects (referenced by global variables), global bindings (included in Smalltalk globals associations) and class variable bindings (included in the classPool of a registered class).

This mapping is done at object granularity, *e.g.,* not every class will be recognized as external. If a class is not in Smalltalk globals or if it has been specified as an *internal class*, it will be traversed and serialized in full detail.

Transient values. There are two main ways of declaring transient values in Fuel. On the one hand, through the hook method fuelIgnoredInstanceVariableNames, where the user specifies variable names whose values will not be traversed nor serialized. On materialization, they will be restored as nil. On the other hand, as we provide the possibility to substitute an object in the actual graph by another one, then an object with transient values can substitute itself by a copy but with such values set to nil. This technique gives a great flexibility to the user for supporting different forms of transient values.

DOI: 10.1002/spe

Class shape change tolerance. Fuel stores the list of variable names of the classes that have instances in the graph being written. While recreating an object from the stream, if its class has changed, then this meta information serves to automatically adapt the stored instances. When an instance variable does not exist anymore, its value is ignored. If an instance variable is new, it is restored as nil. This is true not only for changes in the class but also for changes in any class of the hierarchy. Nevertheless, there are much more kinds of changes a class can suffer that we are not yet able to handle correctly. This is a topic we have to improve.

Versioning. We sign the stream with a well-known string prefix, and then we write the version number of the serializer. Then, when loading the signature and the version has to match with current materializer. Otherwise, we signal an appropriate error. At the moment, we do not support backward compatibility.

4.3. Discussion

Since performance is an important goal for us, we could question why to develop a new serializer instead of optimizing an existing one. For instance, the benefits of using a buffered stream for writing could apply to any serializer. Traditional serializers based on a recursive format commonly implement a technique of caching classes to avoid decoding and fetching the classes on each new instance. The advantages of our clustering format for fast materialization may look similar to such optimization.

Despite of that, we believe that our solution is necessary to get the best performance. The reasons are:

- The caching technique is not as fast as our clustered pickle format. Even if there is cache, the type is *always* written and read per object. Depending of the type of stream, for example, network-based streams, the time spent to read or write the type can be bigger than the time to decode the type and fetch the associated class.
- Since with the cache technique the type is written per object, the resulted stream is much bigger than Fuel's one (since we write the type once per cluster). Having larger streams can be a problem in some scenarios.
- Fuel's performance is not only due to the pickle format. As we explained at the beginning of this section, there are more reasons.

Apart from the performance point of view, there is a set of other facts that makes Fuel valuable in comparison with other serializers:

- It has an object-oriented design, making it easy to adapt and extend to custom user needs. For example, as explained in Section 7, Fuel was successfully customized to correctly support Newspeak modules or proxies in Marea's object graph swapper.
- It can serialize and materialize objects that are usually unsupported in other serializers such as global associations, block closures, contexts, compiled methods, classes and traits. This is hard to implement without a clear design.
- It is modular and extensible. For example, the core functionality to serialize plain objects is at the Fuel package, while another named FuelMetalevel is built of top of it, adding the possibility to serialize classes, methods, traits, etc. Likewise, on top of FuelMetalevel, FuelPackageLoader supports saving and loading complete packages without making use of the compiler.
- It does not need any special support from the VM.
- It is covered by tests and benchmarks.

DOI: 10.1002/spe

5. FUEL DESIGN AND INFRASTRUCTURE

Figure 3 shows a simplified UML class diagram of Fuel's design. It is challenging to explain a design in a couple of lines and a small diagram. However, the following are the most important characteristics about Fuel's design.

- Serializer, Materializer and Analyzer, marked in bold boxes, are the *API* for the whole framework. They are facade classes that provide what most users need to serialize and materialize. In addition they act as builders, creating on each run an instance of Serialization, Materialization and Analysis, respectively, which implement the algorithms. Through *extension methods* we modularly add functionalities to the protocol. For example, the optional package FuelProgressUpdate adds the message showProgress to the mentioned facade classes, which activates a progress bar when processing. We have also experimented with a package named FuelGzip, which adds the message writeGzipped to Serializer, providing the possibility to compress the serialization output.
- The hierarchy of Mapper is an important characteristic of our design: Classes in this hierarchy make possible the complete customization of how the graph is traversed and serialized. They implement the *Chain of Responsibility* design pattern for determining what cluster corresponds to an object [2]. An Analyzer creates a chain of mappers each time we serialize.
- The hierarchy of Cluster has 44 subclasses, where 10 of them are optional optimizations.
- Fuel has 1861 lines of code, split in 70 classes. Average number of methods per class is 7 and the average lines per method is 3.8. We made this measurements on the *core* package Fuel. Fuel is covered by 192 unit tests that covers all use cases. Its test coverage is more than 90% and the lines of code of tests is 1830, almost the same as the core.
- Fuel has a complete benchmark framework which contains samples for all necessary primitive objects apart from samples for large object graphs. We can easily compare serialization and materialization with other serializers. It has been essential for optimizing our performance and verifying how much each change impacts during development. In addition, the tool can export results to csv files which ease the immediate build of charts.

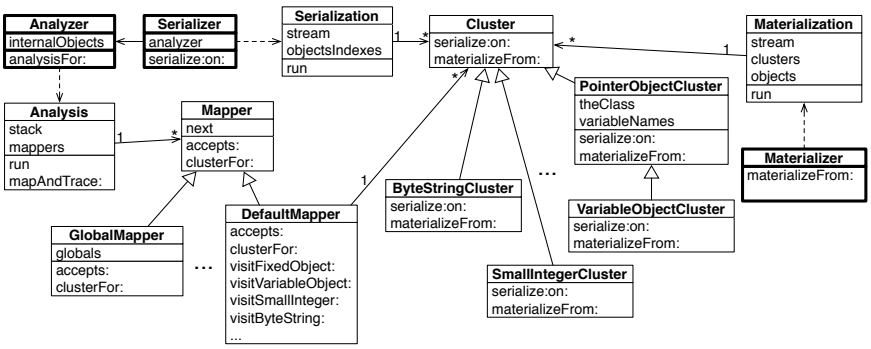

Figure 3. Fuel's design.

6. BENCHMARKS

We have developed several benchmarks to compare different serializers. To get meaningful results all benchmarks have to be run in the same environment. Since Fuel is developed in Pharo, we run all the benchmarks with Pharo-1.3 and Cog Virtual Machine version "VMMaker.oscog-eem.56". The operating system was Mac OS 10.6.7.

DOI: 10.1002/spe

In the following benchmarks, we have analyzed the serializers: Fuel (version 1.7), SIXX (version 0.3.6), SmartRefStream (the version in Pharo 1.3), ImageSegment (the version in Pharo 1.3), Magma object database serializer (version 1.2), StOMP (version 1.8) and SRP (version SRP-mu.11). Such serializers are explained in Section 8.

6.1. Benchmarks constraints and characteristics

Benchmarking software as complex as a serializer is difficult because there are multiple functions to measure which are used independently in various real-world use-cases. Moreover, measuring only the speed of a serializer, is not complete and it may not even be fair if we do not mention the provided features of each serializer. For example, providing a hook for user-defined reinitialization action after materialization or supporting class shape changes slows down serializers. Here is a list of constraints and characteristics we used to get meaningful benchmarks:

All serializers in the same environment. We are not interested in comparing speed with serializers that run in a different environment than Pharo because the results would be meaningless.

Use default configuration for all serializers. Some serializers provide customizations to improve performance, *i.e.,* some parameters or settings that the user can set for serializing a particular object graph. Those settings would make the serialization or materialization faster or slower, depending on the customization. For example, a serializer can provide a way to do *not* detect cycles. Detecting cycles takes time and memory hence, not detecting them is faster. Consequently, if there is a cycle in the object graph to serialize, there will be a loop and finally a system crash. Nevertheless, in certain scenarios, the user may have a graph where he knows that there are no cycles.

Streams. Another important point while measuring serializers performance is which stream will be used. Usually, one can use memory-based streams and file-based streams. There can be significant differences between them and all the serializers must be benchmarked with the same type of stream.

Distinguish serialization from materialization. It makes sense to consider different benchmarks for the serialization and for the materialization.

Different kinds of samples. Benchmark samples are split in two kinds: primitive and large. Samples of primitive objects are samples with lots of objects which are instances of the same class and that class is "primitive". Examples of those classes are Bitmap, Float, SmallInteger, LargePositiveInteger, LargeNegativeInteger, String, Symbol, WideString, Character, ByteArray, etc. Large objects are objects which are composed by other objects which are instances of different classes, generating a large object graph.

Primitive samples are useful to detect whether one serializer is better than the rest while serializing or materializing certain type of object. Large samples are more similar to the expected user provided graphs to serialize and they try to benchmark examples of real life object graphs.

Avoid JIT side effects. In Cog (the VM we used for benchmarks), the first time a method is used, it is executed in the standard way and added to the method cache. The second time the method is executed (when it is found in the cache), Cog converts that method to machine code. However, extra time is needed for such task. Only the third time, the method will be executed as machine code and without extra effort.

It is not fair to run with methods that have been converted to machine code together with methods that have not. Therefore, for the samples, we first run twice the same sample without taking into account its execution time to be sure we are always in the same condition. Then, the sample is finally run and its execution time is computed. We run several times the same sample and take the average of it.

DOI: 10.1002/spe

6.2. *Benchmarks serializing primitive and large objects*

Primitive objects serialization. Figure 4 shows the results of primitive objects serialization and materialization using memory-based streams. The conclusions are:

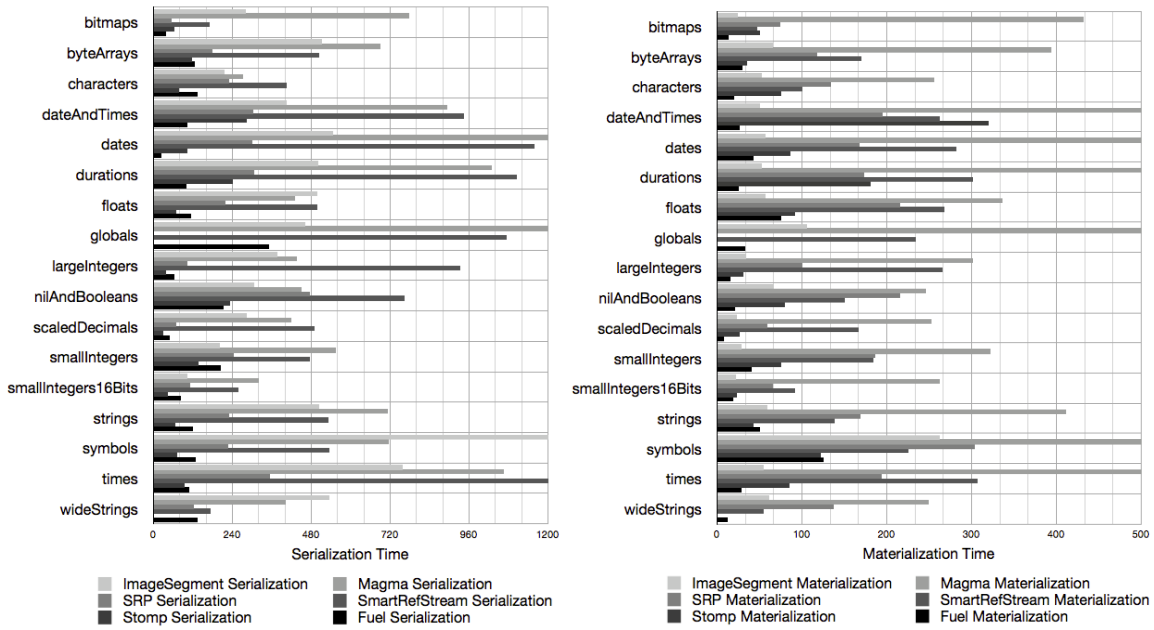

Figure 4. Time (in ms) for primitive objects serialization and materialization (the smaller the better).

- We did not include SIXX in the charts because it was so slow that we were not able to show the differences between the rest of the serializers. This result is expected since SIXX is a text based serializer which is far slower than a binary one. However, SIXX can be opened and modified by any text editor. This is an usual trade-off between text and binary formats.
- Magma and SmartRefStream serializers seem to be the slowest ones in most cases.
- StOMP is the fastest one in serialization nearly followed Fuel, SRP and ImageSegment.
- Magma serializer is slow with "raw bytes" objects such as Bitmap and ByteArray, etc.
- Most of the times, Fuel is faster than ImageSegment, which is even implemented in the Virtual Machine.
- ImageSegment is really slow with Symbol instances. We explain the reason later in Section 6.3.
- StOMP has a zero (its color does not even appear) in the WideString sample. That means that it cannot serialize those objects.

For materialization, *Fuel is the fastest one followed by StOMP and ImageSegment*. In this case and in the following benchmarks, we use memory-based streams instead of file or network ones. This is to be fair with the other serializers. Nonetheless, Fuel does certain optimizations to deal with slow streams like file or network. Basically, it uses an internal buffer that flushes when it is full. This is only necessary because the streams in Pharo are not very performant. This means that, if we run these same benchmarks but using *e.g.,* a file-based stream, Fuel is at least 3 times faster than the second one in serialization. This is important because, in the real uses of a serializer, we do not usually serialize to memory, but to disk.

Large objects serialization. As explained, these samples contain objects which are composed by other objects that are instances of different classes, generating a large object graph. Figure 5 shows the results of large objects serialization and materialization. The conclusions are:

DOI: 10.1002/spe

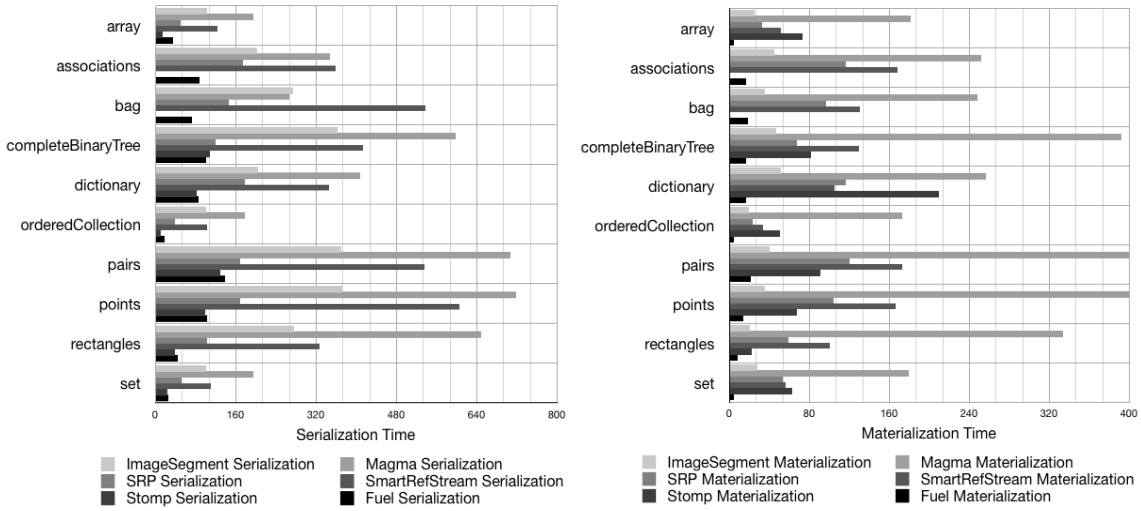

Figure 5. Time (in ms) for large objects serialization and materialization (the smaller the better).

- The differences in speed are similar to the previous benchmarks. This means that, whether we serialize graphs of all primitive objects or objects instances of all different classes, *Fuel is the fastest one in materialization and one of the best ones in serialization*.
- StOMP cannot serialize the samples for *associations* and *bag*. This is because those samples contain different kind of objects, which StOMP cannot serialize. This demonstrates that the mentioned serializers do not support serialization and materialization of all kind of objects. At least, not out-of-the-box. Notice also that all these large objects samples were build so that most serializers do not fail. To have a rich benchmark, we have already excluded different types of objects that some serializers do not support.

6.3. ImageSegment results explained

ImageSegment seems to be really fast in certain scenarios. However, it deserves some explanations of how ImageSegment works [14]. Basically, ImageSegment gets a user defined graph and needs to distinguish between *shared objects* and *inner objects*. Inner objects are those inside the subgraph which are *only* referenced from objects inside the subgraph. *Shared objects* are those which are not only referenced from objects inside the subgraph, but also from objects outside.

All *inner objects* are put into a byte array which is finally written into the stream using a primitive implemented in the virtual machine. Afterwards, ImageSegment uses SmartRefStream to serialize the *shared objects*. ImageSegment is fast mostly because it is implemented in the virtual machine. However, as we saw in our benchmarks, SmartRefStream is not really fast. The real problem is that it is difficult to control which objects in the system are pointing to objects inside the subgraph. Hence, there are frequently several *shared objects* in the graph. The result is that, the more *shared objects* there are, the slower ImageSegment is because those *shared objects* will be serialized by SmartRefStream.

All the benchmarks we did with primitive objects (all but Symbol) create graphs with zero or few shared objects. This means that we are measuring the fastest possible case ever for ImageSegment. Nevertheless, in the sample of Symbol, one can see in Figure 4 that ImageSegment is really slow in serialization and the same happens with materialization. The reason is that, in Smalltalk, all instances of Symbol are unique and referenced by a global table. Hence, all Symbol instances are shared and, therefore, serialized with SmartRefStream.

Figure 6 shows an experiment we did where we build an object graph and we increase the percentage of *shared objects*. Axis *X* represents the percentage of shared objects inside the graph and the axis *Y* represents the time of the serialization or materialization.

DOI: 10.1002/spe

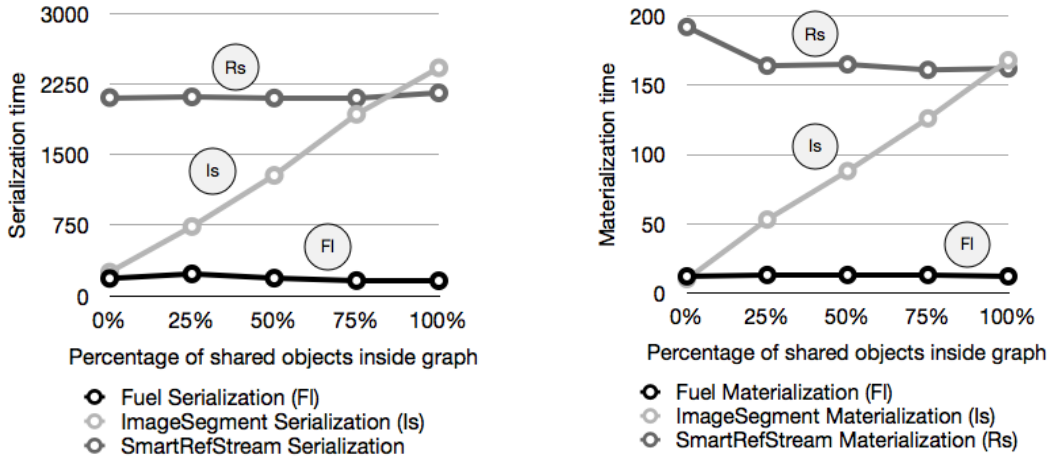

Figure 6. ImageSegment serialization and materialization in presence of shared objects.

Conclusions for ImageSegment results

- The more shared objects there are, the more similar is ImageSegment speed compared to SmartRefStream.
- For materialization, when all objects are shared, ImageSegment and SmartRefStream have almost the same speed.
- For serialization, when all objects are share, ImageSegment is even slower than SmartRefStream. This is because ImageSegment needs to do the whole memory traverse anyway to discover shared objects.
- ImageSegment is unique in the sense that its performance depends on both: 1) the amount of references from outside the subgraph to objects inside; 2) the total amount of objects in the system since the time to traverse the whole memory depends on that.

6.4. Different graph sizes

Another important analysis is to determine if there are differences between the serializers depending on the size of the graph to serialize. We created different subgraphs of different sizes. To simplify the charts we express the results in terms of the largest subgraph size which is 50.000 objects. The scale is expressed as percentage of this size.

Figure 7 shows the results of the experiment. Axis *X* represents the size of the graph, which in this case is represented as a percentage of the largest graph. Axis *Y* represents the time of the serialization or materialization.

Conclusions for different graph sizes. There are not any special conclusions for this benchmark since the performance differences between the serializers are almost the same with different graph sizes. In general the serializers have a linear dependency with the number of objects of the graph. For materialization, Fuel is the fastest and for serialization is similar than StOMP. Fuel performs then well with small graphs as well as large ones.

6.5. Differences while using CogVM

At the beginning of this section, we explained that all benchmarks are run with the Cog Virtual Machine. Such a virtual machine introduces several significant improvements such as PIC (polymorphic inline cache) and JIT (just in time compiling) which generates machine code from interpreted methods. All those improvements impact, mainly, in regular methods implemented in the language side but not in VM inside itself, VM primitives or plugins.

DOI: 10.1002/spe

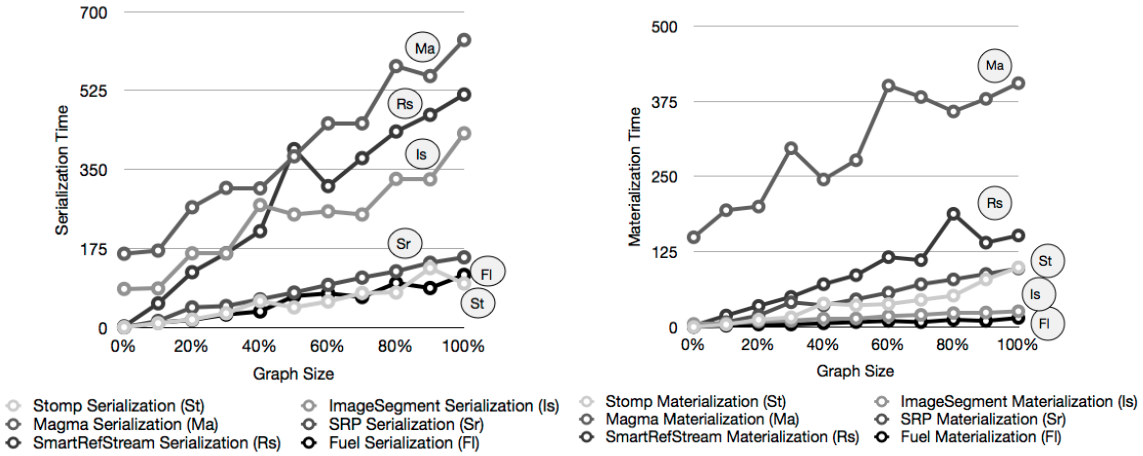

Figure 7. Serialization and materialization of different graph sizes.

Cog is at least 4 times faster than the interpreter VM (the previous VM). It is a common belief that ImageSegment was the fastest serialization approach. However, along this section, we showed that Fuel is most of the times as fast as ImageSegment and sometimes even faster.

One of the reasons is that, since ImageSegment is implemented as VM primitives and Fuel is implemented in the language side, with Cog Fuel, the speed increases four times while ImageSegment speed remains almost the same. This speed increase takes place not only with Fuel, but also with the rest of the serializers implemented in the language side.

To demonstrate these differences, we did an experiment: we ran the same benchmarks with ImageSegment and Fuel with both virtual machines, Cog and Interpreter VM. For each serializer, we calculated the difference in time of running both virtual machines. Figure 8 shows the results of the experiment. Axis *X* represents the difference in time between running the benchmarks with Cog and non Cog VMs.

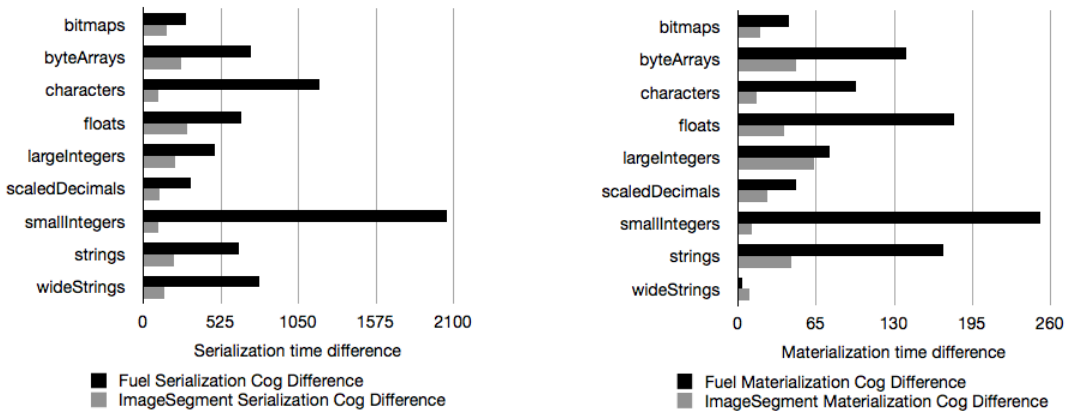

Figure 8. Serialization and materialization differences when using CogVM

As we can see in both operations (serialization and materialization), the difference in Fuel is much bigger that the difference in ImageSegment.

Prepared using speauth.cls
DOI: 10.1002/spe

6.6. General benchmarks conclusions

Magma serializer seems slow but it is acceptable taking into account that this serializer is designed for a particular database. Hence, the Magma serializer does an extra effort and stores extra information that is needed in a database scenario but may not be necessary for any other usage.

SmartRefSteam provides a good set of hook methods for customizing serialization and materialization. However, it is slow and its code is complex and difficult to maintain from our point of view. ImageSegment is known to be really fast because it is implemented inside the virtual machine. Such fact, together with the problem of *shared objects*, brings a large number of limitations and drawbacks as it has been already explained. Furthermore, with Cog, we demonstrate that Fuel is even faster in both, materialization and serialization. Hence, the limitations of ImageSegment are not worth it.

SRP and StOMP are both aimed for portability across Smalltalk dialects. Their performance is good, mostly at writing time, but they are not as fast as they could be because of the need of being portable across platforms. In addition, for the same reason, they do not support serialization for all kind of objects.

This paper demonstrates that Fuel is the fastest in materialization and one of the fastest ones in serialization. In fact, when serializing to files, which is what usually happens, Fuel is the fastest. Fuel can also serialize any kind of object. Fuel aim is not portability but performance. Hence, all the results make sense from the goals point of view.

7. REAL CASES USING FUEL

Even if Fuel is still in development, it is already being used in real applications. Here we report the first ones we are aware of. Fuel is also ported by external developers to other Smalltalk dialects.

Moose Models. Moose is an open-source platform for software and data analysis [18]. It uses large data models that can be exported and imported from files. The models produces by Moose represent source code and information produced by analyzers. A Moose model can easily contain $500,000$ entities. Moose is also implemented on top of Pharo. The Fuel Moose extension is eight times faster in exporting and four times faster in importing than its competitor MSE. We have developed a model export/import extension[†] which has been integrated into Moose Suite 4.4.

Pier CMS persistency. Pier is a content management system that is light, flexible and free [22]. It is implemented on top of the Seaside web framework [24]. In Pier all the documents and applications are represented as objects. A simple persistency strategy has been implemented using Fuel[‡]. This tool persists Pier CMS kernels (large object graphs), *i.e.,* Pier CMS websites. This way you can backup your system and load it back later on if desired.

SandstoneDB with Fuel backend. SandstoneDB[§] is a lightweight Prevayler style embedded object database with an ActiveRecord API that does not require a command pattern and works for small applications that a single Pharo image can handle. The idea is to make a Pharo image durable, crash proof and suitable for use in small office applications. By default, SandtoneDB used the SmartRefStream serializer. Now there is a Fuel backend which accelerates SandstoneDB 300% approximately.

Newspeak port. Newspeak[¶] is a language derived from Smalltalk. Its developers have successfully finished a port of Fuel to Newspeak and they are using it to save and restore their data sets. They had to implement one extension to save and restore Newspeak classes, which is

[†] http://www.moosetechnology.org/tools/fuel
[‡] http://ss3.gemstone.com/ss/pierfuel.html
[§] http://onsmalltalk.com/sandstonedb-simple-activerecord-style-persistence-in-squeak
[¶] http://newspeaklanguage.org/

Prepared using speauth.cls
DOI: 10.1002/spe

complex because these are instantiated classes inside instantiated Newspeak modules [6] and not static Smalltalk classes in the Smalltalk dictionary. Fuel proved to be flexible enough to make such port successful taking only few hours of work.

Marea. Marea is a transparent object graph swapper whose goal is to use less memory by only leaving in primary memory what is needed and used serializing and swapping out the unused objects to secondary memory [15, 16]. When one of these unused objects is needed, Marea brings it back into primary memory. To achieve this, the system replaces original objects with proxies. Whenever a proxy receives a message, it loads back and materializes the swapped out object from secondary memory. Marea needs to correctly serialize and materialize any type of object such as classes, methods, contexts, closures, etc.

When Fuel is processing a graph, it sends messages to each object, such as asking its state or asking its hash (to temporally store it into a hashed collection). But in presence of proxies inside the graph, that means that the proxy intercepts those messages and swaps in the swapped out graph. To solve this problem, Marea extends Fuel so that it takes special care when sending messages to proxies. As a result, Marea can serialize graphs that contain proxies without causing them to swap in.

8. RELATED WORK

We faced a general problem to write a decent related work: serializers are not clearly described. At the best, we could execute them, sometimes after porting effort. Most of the times there was not even documentation. The rare work on serializers that we could find in the literature was done to advocate that using C support was important [23]. But since this work is implemented in Java we could compare and draw any scientific conclusion.

The most common example of a serializer is one based on XML like SIXX [25] or JSON [12]. In this case, the object graph is exported into a portable text file. The main problem with text-based serialization is encountered with big graphs as it does not have a good performance and it generates huge files. Other alternatives are ReferenceStream or SmartReferenceStream. ReferenceStream is a way of serializing a tree of objects into a binary file. A ReferenceStream can store one or more objects in a persistent form including sharing and cycles. The main problem of ReferenceStream is that it is slow for large graphs.

A much more elaborated approach is Parcel [17] developed in VisualWorks Smalltalk. Fuel is based on Parcel's pickling ideas. Parcel is an atomic deployment mechanism for objects and source code that supports shape changing of classes, method addition, method replacement and partial loading. The key to making this deployment mechanism feasible and fast is the pickling algorithm. Although Parcel supports code and objects, it is more intended to source code than normal objects. It defines a custom format and generates binary files. Parcel has a good performance and the assumption is that the user may not have a problem if saving code takes more time as long as loading is really fast. The main difference with Parcels is that such project was mainly for managing code: classes, methods, and source code. Their focus was that, and not to be a general-purpose serializer. Hence, they deal with problems such as source code in methods, or what happens if we install a parcel and then we want to uninstall it, what happened with the code, and the classes, etc. Parcel is implemented in Cincom Smalltalk so we could not measure their performance to compare with Fuel.

The recent StOMP [27] (Smalltalk Objects on MessagePack[‖]) and the mature SRP (State Replication Protocol) [26] are binary serializers with similar goals: Smalltalk-dialect portability and space efficiency. They are quite fast and configurable but they are limited with dialect-dependent objects like BlockClosure and MethodContext.

[‖] http://msgpack.org

DOI: 10.1002/spe

Object serializers are needed and used not only by final users, but also for specific type of applications or tools. What is interesting is that they can be used outside the scope of their project. Some examples are the object serializers of Monticello2 (a source code version system), Magma object database, Hessian binary web service protocol [10] or Oracle Coherence*Web HTTP session management [19].

Martinez-Peck et al. [14] performed an analysis of ImageSegment (a virtual machine serialization algorithm) and they found that the speed increase in ImageSegment is mainly because it is written in C compared to other frameworks written in Smalltalk. However, ImageSegment is slower when objects in the subgraph to be serialized are externally referenced.

9. CONCLUSION AND FUTURE WORK

In this paper, we have looked into the problem of serializing object graphs in object-oriented systems. We have analyzed its problems and challenges, which are general and independent of the technology.

These object graphs operations are important to support virtual memory, backups, migrations, exportations, etc. Speed is the biggest constraint in these kind of graph operations. Any possible solution has to be fast enough to be actually useful. In addition, the problem of performance is the most common one among the different solutions. Most of them do not deal properly with it.

We presented Fuel, a general purpose object graph serializer based on a pickling format and algorithm different from typical serializers. The advantage is that the unpickling process is optimized. On the one hand, the objects of a particular class are instantiated in bulk since they were carefully sorted when pickling. This is done in an iterative instead of a recursive way, which is what most serializers do. The disadvantage is that the pickling process takes extra time in comparison with other approaches. However, we show in detailed benchmarks that we have the best performance in most of the scenarios.

We implement and validate this approach in Pharo. We demonstrate that it is possible to build a fast serializer without specific VM support with a clean object-oriented design and providing the most possible required features for a serializer.

Instead of throwing an error, it is our plan to analyze the possibility of creating light-weight shadow classes when materializing instances of an inexistent class. Another important issue we would like to work on is in supporting backward compatibility and migration between different Fuel versions. Partial loading as well as the possibility of being able to query a serialized graph, are two concepts that are in our roadmap.

To conclude, Fuel is a fast object serializer built with a clean design, easy to extend and customize. New features will be added in the future and several tools will be build on top of it.

ACKNOWLEDGEMENT

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the CPER 2007-2013.

REFERENCES

1. Action message format - amf 3. http://download.macromedia.com/pub/labs/amf/amf3_spec_121207.pdf.
2. Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.
3. John K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 318–330, December 1987.
4. Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.
5. Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In Gail E. Harris, editor, *OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems,*

DOI: 10.1002/spe

Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA, pages 109–126. ACM, 2008.
6. Gilad Bracha, Peter von der Ahé, Vassili Bykov3, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In *ECOOP 2009*, LNCS. Springer, 2009.
7. Fabian Breg and Constantine D. Polychronopoulos. Java virtual machine support for object serialization. In *Joint ACM Java Grande - ISCOPE 2001 Conference*, 2001.
8. Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.
9. Dominique Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 444–452, November 1986.
10. Hessian. http://hessian.caucho.com.
11. Java serializer api. http://java.sun.com/developer/technicalArticles/Programming/serialization/.
12. Json (javascript object notation). http://www.json.org.
13. Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):87–106, November 1986.
14. Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Experiments with a fast object swapper. In *Smalltalks 2010*, 2010.
15. Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Problems and challenges when building a manager for unused objects. In *Proceedings of Smalltalks 2011 International Workshop*, Bernal, Buenos Aires, Argentina, 2011.
16. Mariano Martinez Peck, Noury Bouraqadi, Stéphane Ducasse, and Luc Fabresse. Object swapping challenges: an evaluation of imagesegment. *Journal of Computer Languages, Systems and Structures*, 38(1):1–15, November 2012.
17. Eliot Miranda, David Leibs, and Roel Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Journal of Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.
18. Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.
19. Oracle coherence. http://coherence.oracle.com.
20. Pickle. http://docs.python.org/library/pickle.html.
21. Google protocol buffers. http://code.google.com/apis/protocolbuffers/docs/overview.html.
22. Lukas Renggli. Pier — the meta-described content management system. European Smalltalk User Group Innovation Technology Award, August 2007. Won the 3rd prize.
23. Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, 1996.
24. Seaside home page. http://www.seaside.st.
25. Sixx (smalltalk instance exchange in xml). http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html.
26. State replication protocol framework. http://sourceforge.net/projects/srp/.
27. Stomp - smalltalk objects on messagepack. http://www.squeaksource.com/STOMP.html.
28. David Ungar. Annotating objects for transport to other worlds. In *Proceedings OOPSLA '95*, pages 73–87, 1995.
29. Douglas Wiebe. A distributed repository for immutable persistent objects. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 453–465, November 1986.

DOI: 10.1002/spe