Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Haochuan Feng April 14, 2022

PADL: Persistent Application Data Library

By

Haochuan Feng

Dorian Arnold, Ph.D.
Advisor

Computer Science

Dorian Arnold, Ph.D.
Advisor

Michelangelo Grigni, Ph.D.
Committee Member

Angela Porcarelli, Ph.D.
Committee Member

2022

PADL: Persistent Application Data Library

By

Haochuan Feng

Dorian Arnold, Ph.D.
Advisor

An abstract of
a thesis submitted to the Faculty of the
Emory College of Arts and Sciences of Emory University
in partial fulfillment of the requirements for the degree of
Bachelor of Science with Honors

Computer Science

2022

Abstract

PADL: Persistent Application Data Library
By Haochuan Feng

This thesis presents a new application programming interface (API) that implements an application-level data serialization for C. With the API, users can register or unregister the key variables they want to serialize. At key points of interest during the program's execution, for example, to take a checkpoint that mitigates future hardware or software failures, users can automatically serialize all registered variables. Later the user can retrieve and deserialize previously stored data objects. We show how the API makes it easier for users to implement checkpointing and can make the process more efficient in space and time.

PADL: Persistent Application Data Library

By

Haochuan Feng

Dorian Arnold, Ph.D.
Advisor

A thesis submitted to the Faculty of the
Emory College of Arts and Sciences of Emory University
in partial fulfillment of the requirements for the degree of
Bachelor of Science with Honors

Computer Science

2022

Acknowledgments

First and foremost, I thank my advisor, Professor Dorian Arnold. I took Dr. Arnold's System Programming class in my senior year and became very interested in this field. This was my first experience doing research in computer science. Dr. Arnold gave me a lot of guidance and support throughout the project.

In addition, I thank my committee members, Professor Michelangelo Grigni and Professor Angela Porcarelli. They gave great suggestions for my thesis and consistent support throughout my undergraduate studies at Emory.

Last but not least, I thank my parents who gave me great support from a distance during the special time of COVID-19.

Contents

List of Tables

Chapter 1

Introduction

For many computer programs with long running times, it may be unavoidable to encounter software or hardware faults. If the program must start all over again at each failure, hours, days or even weeks of processing time are wasted. To run to completion, these programs must tolerate such failures in some way.

Checkpoint Recovery is an effective fault tolerant mechanism. Checkpoints save current program state to persistent storage periodically. At system failures, checkpoints can be retrieved to restore the last saved state of the program. There are two main methods for implementing Checkpoint Recovery: system-level and application-level checkpointing.

System-level checkpointing schemes are a "core dump" style of copying the entire data from memory into the hardware. The entire saving and recovery process is a black box to the user and requires no effort from the programmer to implement. Although easy to use, there are two issues with system-level checkpointing. First, system-level checkpointing lacks portability. The checkpoint data can be very system-specific. Thus, the checkpoint file can only be restarted on the same computer. Secondly, taking a global snapshot using system-level checkpointing may include many unnecessary data, for example temporary data, and thus greatly slow

down the running program.

On the other hand, many programmers choose to implement application-level checkpointing. In this way, they control what to save in checkpointing to reduce the amount of data to be included in a checkpoint. Although more efficient in run time, self-implemented checkpointing requires lots of programmer effort. At each checkpoint, the users must decide what to save and implement the save and restore mechanism.

We propose an application-level data serialization library to support checkpointing for C. We call our library Persistent Application Data Library, PADL. With PADL, the users only need to specify the data objects they want to save and when to save them. PADL automatically serializes the data specified in a matter that allows the data to be used by the program on another machine. Using PADL is straightforward, users only need to add variables to registry and remove it from the registry when it is no longer necessary to keep a record of them. Using a prototype implementation, we show that using PADL to create application-level checkpointing is not only easy to implement but also efficient in running time and storage space.

For the rest of this thesis, Chapter 2 describes current serialization methods and their advantages and disadvantages. Chapter 3 describes PADL implementation details. Chapter 4 and 5 present our experimental results and opportunities for future improvements.

Chapter 2

Background and Related Work

The critical step in checkpointing is serialization and deserialization. Serialization is a process to convert objects into a data stream, while deserialization reverses that process. Checkpointing serializes the relevant data into a file that can be deserialized during a failure recovery process. There are many existing serialization libraries targeting different purposes. As a result, they are also very different in their functionality and efficiency in serializing the data.

In this work, the following requirements are essential:

- **Small file size:** The file storage format should be as efficient as possible. Storage overhead can be computed as the percentage of (checkpoint size - data size)/data size.

- **Low serialization overhead:** The time it takes to serialize and deserialize the data should be as low as possible. This is computed as: the time to pack the data + the time to write or read.

- **High programmer productivity:** The interface should be intuitive and easy to use. The amount of programming effort needed to use the API should be low.

- **Portability:** The serialized data stream must contain all information regarding the data, so the data stored can be used to recover for another machine.

Related Data Serialization Works

Several serialization libraries exist on different platforms for various purposes such as communication, synchronization, or cloning process. Some encode objects into very compact bytes streams, while some contain a lot of external information about the objects. Some libraries provide simple functions to write into and read from a byte stream, while others require the users to write a complete data scheme for serialization. Because of the difference in the serialized data stream format and design of library functions, these serialization methods have their advantages and disadvantages. This thesis will compare the four most popular serialization methods: Java serializations, Pickle, FlatBuffer, and Protocol buffer.

Java Serialization [4] can serialize all objects types that have implemented a *serializable* or *externalizable* interface. Because serializable is an interface, users have the flexibility to overwrite the object serialization method and, therefore, to decide which class fields to serialize. The interface can be straightforward to use: users only need to put the variables they want to serialize in an interface and use function *writeObject* and *readObject* to write into and read from a byte stream. The time and space efficiencies are a function of the efficiencies of the user's implementations.

Pickle [5] is a Python-specific library that supports the serialization/deserialization of almost all Python data types into a binary data stream. Pickle is space efficient because it serializes complex objects into a stream in a compact stream. Pickle keeps track of objects already serialized and also supports compression when needed. Pickle also has low serialization/deserialization overhead because encoding only takes place for large objects that needed to be compressed. It is easy to use pickle. You can simply use the functions *dump* and *load* to serialize and deserialize an object, but the user must track the order in which objects are serialized.

FlatBuffer [7] converts data into a byte stream that contains the descriptive metadata and the real data. Because FlatBuffer stores object in a very simple and compact way, it has both small file size and low serialization/deserialization overhead. But it is difficult to implement as users have to write a schema to implement it. The program also needs to use the schema to interpret the byte stream.

Protocol buffer [3] is similar to FlatBuffer. It uses message as a fundamental element. In serialization, message will be serialized into a very compact binary stream. Protocol has small file size. But the serialization overhead is high because of the encoding. Like FlatBuffer, users must specify everything in the message for serialization. Protocol buffer's advantage over FlatBuffer is that there is not need to deserialize the entire data stream before accessing a single object.

In addition to serialization libraries, there are also data serialization format specifications that describe a data format and structure for object serialization/deserialization. Such specifications include XDR, JSON, EXI and HDF5.

XDR [2] is an internet communication protocol that specifies a representation for most commonly used data types in high-level languages, including arrays, structures, and user-defined data types. It does not specify the file format for storage.

JSON [1] is a markup language that uses tags to define elements in a format that is readable for humans. Because of the tags and conversion to readable format, there is a lot of overhead in both storage space and time used for serialization.

XML [8] is a markup language for internet communication. While efficient XML is a more compact way of representation. An EXI stream is self-describing but contains a lot of external information.

HDF5 [6] is a file system designed to store and organize large amounts of data. HDF5 puts the data in a search-able structure. When the user try to search for an object a large data set, the file structure make it faster by reduce the number of time it takes to access the disk. HDF5 is designed for fast access within large data volumes.

The complex file structure will also increase the file size and serialization time. It can be complicated and lengthy to implement HDF5 in C at the primary level. But there are many wrappers on the internet to make it much more manageable.

As shown in Table 2.1, our aim is to close the performance or functionality gaps in existing libraries with PADL, that hopes to meet all our requirements.

	Performance			Functionality	
Name	Small file size	Low serialization / deserialization overhead	Higher programmer productivity	Portability	Language supported
Java serialization	Yes	No	Yes	Yes	Java
Pickle	Yes	No	Yes	Yes	Python
Flatbuffer	Yes	No	Yes	Yes	C++, #, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust, Swift.
Protocol buffer	No	No	No	Yes	C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust, Swift.
PADL	Yes	Yes	Yes	Yes	C

Table 2.1: Summary of the Features of Popular Data Serialization Libraries.

Chapter 3

PADL Implementation

PADL uses an in-memory register table to track objects for serialization. Users must first call an initialization function, which initializes all global variables, creates the register table in memory, creates a checkpoint directory to keep serialized data files and creates an information file in the checkpointing directory to store the general checkpointing information. Then the user can register or unregister important objects for subsequent serialization. When serialization is desired, the user can call a serialization function that serializes all variables in the register table into a data file in the checkpoint directory. If the number of serialized data files exceed a user-controlled limit, the earliest data files in the checkpoint directory are deleted. When desirable or necessary, the user can use PADL's deserialization function to extract object values from the data file and put them back into the variables stored in the register table. Upon program completion, the user can call a PADL termination function to erase all memory traces and delete the directory and all the serialization files.

3.1 PADL Core Functions

The core functions implement the main PADL functionalities. All core functions return 0 when successfully executed and a number associated with the encountered

error, otherwise. (Details of the error numbers and error messages are in Appendix 5.)

```
int initialize (char * name, unsigned int maxNumofVersions)
```

This function creates the global variables, a register table, a checkpoint directory of the name given, and an information file, "InfoFile", inside the checkpoint directory. If a directory of the same name already exists. The info file will be validated for the correct format and updated. The user-controlled limit of the number of data files is set as maxNumofVersions.

```
int register (char * varName, char * type, void * address, unsigned int size)
```

This function adds information about an initialized variable into the register table and labels the variable as specified by *varName*. The register table keeps a record of the variable's name, type, address, and array size (1 if it is a single variable) as provided in the input of the function.

```
void unregister (char * varName)
```

This function removes the variable, *varName* from the register table.

```
void serialize ()
```

This function serializes all *enabled* variables in the register table. The serialized data stream will be stored in a Data file of the named, "Data" + the current version number. The earliest data file will be deleted if the number of data files exceeds the user-defined limit.

```
void deserialize (unsigned int fileNum)
```

This function deserializes all registered variables from the data in the checkpointing files. That is, the registered variable data specifies the addresses to which retrieved variables should be written.

```
void terminate (int keepData)
```

This function erases all PADL-related in-memory data. If *keepData* equals 1, the checkpointing files will be kept, otherwise, they will be deleted.

3.2 PADL Usage Demonstration

In Figure 3.1 is a simple demonstration code of how to use PADL. First, initialize() is called to create a checkpointing called "MyCheckpoints" and to limit the number of serialized data files to 10. Three variables: variable1, variable2, and variable3, are initialized with type int, double, and char and put into the register with their names according to their types (%d, %c, and %lf), address, and array size using register(). (Since variable1 is just a single variable, the array size is 1.) Later, serialize() is called to serialize all three variables into a data file. When serialized is called again, the same three variables are serialized into another data file. When it is no longer necessary to save variable1, variable1 can be removed from the register table using unregister(). In the subsequent serialization, only variable2 and variable3 will be serialized. Upon program completion, function terminate is called with input 0 to delete all PADL in-memory data and files generated during the checkpointing.

3.3 Auxiliary PADL Functions

In addtion to the core functions in the previous section, PADL has other necessary support functions as well as some convenience functions. For instance, errorMessage() takes the error number as input and returns the error number's respective description. validate() checks whether the info file and data file are of the correct format and complete. The function listVar() lists the names of all the variables in the data file. Full details of the complete PADL API is in Appendix 5.

```
initialize("MyCheckpoints", 10);
int variable1 = 1;
char * variable2 = malloc(sizeof(char) * 100);
double * varialbe3 = malloc(sizeof(double) * 20);

register("variable1", "\%d", \&variable1, 1);
register("variable2", "\%c", variable2, 100);
register("variable3", "\%lf", variable3, 20);
...

serialize();
...

serialize();
...

unregister("variable1");
serialize();
...

terminate(0);
```

Figure 3.1: Caption

3.4 Supported Data Types

Table 3.1 presents the data types supported by PADL. It covers all the basic data types in C, including the basic types character, integer, and float-point number, with the type specifiers c, d, and f. u, h and l are special modifiers meaning unsigned, short, and long. In our current implementation, for more complex data types such as structure and pointer of pointers, users can use the byte stream and specify the total size of the object. But at serialization and deserialization, the users must wrap and unwrap the variable by themselves.

3.5 File Format

Tables 3.2 and 3.3 show the expected format for the $InfoFile$, which stores general metadata, and checkpoint files, which store the actual serialized data objects. The

Short Notation	Description
c	signed 8-bit character
uc	unsigned 8-bit character
hd	signed 16-bit decimal integer
uhd	unsigned 16-bit decimal integer
d	signed 32-bit decimal integer
ud	unsigned 32-bit decimal integer
ld	signed 64-bit decimal integer
uld	unsigned 64-bit decimal integer
f	32-bit floating-point number
lf	64-bit floating-point number
b	byte stream

Table 3.1: Supported Data Types

latter files are named $Data - \%n$, where %n is the checkpointing version number. (Both information and data files start with a special magic number that matches their expected format.)

Bytes	Data
8	InfoFile Magic Number
4 (unsigned int)	Size of Address
4 (unsigned int)	Endianness (0 big, 1 small)
4 (unsigned int)	Max Version Number Keep
4 (unsigned int)	Current Number of Versions

Table 3.2: Info File(Name: "InfoFile")

Bytes	**Data**
	Data Content
8	Next var location in file
4 (unsigned int)	Len (name)
4 (unsigned int)	Len (type)
4 (unsigned int)	Size
Len (name)	Name
Len (type)	Type
Size * Unit Size	Object Value
8	End Signature

Table 3.3: Data File (Name: "Data" + versionNum)

Chapter 4

Performance Test

In this section, we present the results of the performance of our PADL protoype implementation. We evaluated two main performance features: data file space efficiency and serialization overhead. The API complexity is also of interest, but in this work we do not quantitatively evaluate interface complexity.

4.1 Experimental Setup

Our test program consists of all ten basic types in C and arrays of these types of lengths 1, 10, 50, 100, 500, 2000, and 5000. Therefore, the total length of objects for each array length is: 42 bytes, 420 bytes, 2100 bytes, 4200 bytes, 21000 bytes, 84000 bytes, and 210000 bytes. We report the data file lengths, percentage of data file overheads ($checkpoint_size - data_size : datasize$), serialization times and deserialization times. We measure deserialization times both on machines of similar and different endianness.

The experiment were performed on a MacBook Pro with 2.6 GHz 6-Core Intel Core i7 processor and 16 GB memory size.

4.2 Experimental Results

	Space Efficiency in Bytes		Serialization Overhead in Seconds (1000 times)		
Data	Data file size	Data File Space overhead	Serialization	Deserialization	Different Endianness Deserialization
All Variables	323,680	1%	1.072455	0.110896	0.283054
1 Array	5,279	99%	0.270415	0.048327	0.045063
10 Array	5,657	93%	0.259049	0.108694	0.103219
50 Array	7,337	71%	0.261118	0.097684	0.097017
100 Array	9,437	55%	0.266048	0.051867	0.053977
500 Array	26,237	20%	0.310437	0.047437	0.044668
2000 Array	89,237	6%	0.459381	0.049553	0.049744
5000 Array	210,218	1%	0.776523	0.047976	0.044700

Table 4.1: Experimental Results

4.3 Conclusion

The test results, in Table 4.1, show that the API can be very fast in implementation for serialization and deserialization, even for a relatively large number of variables. The file space overhead is considerable for objects of small sizes because a similar amount of space is needed to store related information to deserialize for both large and small objects. The absolute amount of data for an object's metadata is fixed at about 20 bytes. This is not a problem, given that the number of variables in a program is generally limited, and the data objects are typically very large. In conclusion, we believe PADL not only easy to use but also very efficient in running time and storage space.

Chapter 5

Future Enhancement

There are several opportunities to extend this work:

- Support for aggregate data types: currently, PADL only supports the basic data types and arrays in C, but aggregate data types such as structure and pointer of pointers are essential features in C.

- Smaller data files: mechanisms like compression and tracking duplicate objects can be used to further improve both the size of PADL data files and reduce serialization overhead. PADL could also consider incremental updates that only save parts of a previously-saved data object that has changed.

- Comprehensive evaluation: a more comprehensive evaluation will be useful to (1) understand PADL's overhead in more realistic contexts, (2) separate the overhead of PADL serialization from the cost of I/O transfers or storage device overheads, and (3) quantitatively compare PADL to other data serialization libraries.

Bibliography

[1] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017. URL https://www.rfc-editor.org/info/rfc8259.

[2] Mike Eisler. XDR: External Data Representation Standard. RFC 4506, May 2006. URL https://www.rfc-editor.org/info/rfc4506.

[3] Google. Protocol buffers version 3 language specification, 2022. URL https://developers.google.com/protocol-buffers/docs/reference/proto3-spec.

[4] Oracle. Java object serialization specification, 2005. URL https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html.

[5] Python Software Foundation. pickle — Python object serialization, 2022. URL https://docs.python.org/3/library/pickle.html.

[6] The HDF Group. Hierarchical data format version 5, 2000-2022. URL https://portal.hdfgroup.org/display/HDF5/HDF5.

[7] Wouter van Oortmerssen. Flatbuffers. URL https://google.github.io/flatbuffers/flatbuffers_white_paper.html.

[8] Efficient XML Interchange (EXI) WG. Efficient xml interchange (exi) format 1.0 (second edition), 2014.

PADL Error Messages

Error Number	**Error Message**
1	"Directory of the same name already existed."
2	"Failure at creating the directory."
3	"Failure at creating the file."
4	"Variable containing null pointers."
5	"Variable type unrecognized."
6	"Missing Info File."
7	"Missing Data file."
8	"Info File has incorrect format signature."
9	"Variable is not found in the registry."
10	"Info file has the wrong format."
11	"Checkpointing file has the wrong format."
12	"Checkpointing file is incomplete."
13	"Variable has different allocated sizes."

Table 1: Error Message

Support Functions

```
char * errorMessage(int i)
```

This function takes the error number and returns the error message in a string.

```
int disableVar(char* varName)
```

This function changes the state of the variable, *varName* to *disabled*.

```
int enableVar(char* varName)
```

This function changes the state of the variable, *varName* to *enabled*.

```
int validate(unsigned int fileNum)
```

This function checks whether the info file and data file are of the correct format and complete.

```
int getCheckpointNum()
```

This function returns the latest checkpointing version number.

```
VarInfo getVarInfo(unsigned int fileNum, char* VarName)
```

This function returns the name, type, and size of variable, *varName* recorded in the data file.

```
char**  listVar(unsigned int fileNum)
```

This function returns the names of all the variables in the data file as an array of strings.

```
int varExist(unsigned int fileNum, char* varName)
```

This function returns 1 if the variable exists in the data file, 0 otherwise.

```
int varState(char* varName)
```

This function returns the state of the variable inside the register: -1 if not found, 0 if disabled, 1 if enabled.