MiDas: Containerizing Data-Intensive Applications with I/O Specialization

Chaitra Niddodi
University of Illinois at Urbana-Champaign
chaitra@illinois.edu

Ashish Gehani
SRI International
ashish.gehani@sri.com

Tanu Malik
DePaul University
tanu@cdm.depaul.edu

Jorge A. Navas
SRI International
jorge.navas@sri.com

Sibin Mohan
University of Illinois at Urbana-Champaign
sibin@illinois.edu

ABSTRACT

Scientific applications often depend on data produced from computational models. Model-generated data can be prohibitively large. Current mechanisms for sharing and distributing reproducible applications, such as *containers*, assume all model data is saved and included with a program to support its successful re-execution. However, including model data increases the sizes of containers. This increases the cost and time required for deployment and further reuse. We present a framework named **MiDas** ("Minimizing Datasets") for *specializing* I/O libraries which, given an application, automates the process of identifying and including only a subset of the data accessed by the program. To do this, **MiDas** combines static and dynamic analysis techniques to map high level user inputs to low level file offsets. We show several orders of magnitude reduction in data size via specialization of I/O libraries associated with model-based data-intensive applications, such as those operating on meteorological and geophysical data.

CCS CONCEPTS

• **Information systems** → **Specialized information retrieval**.

KEYWORDS

Containers, Data-Intensive, I/O specialization

ACM Reference Format:
Chaitra Niddodi, Ashish Gehani, Tanu Malik, Jorge A. Navas, and Sibin Mohan. 2020. MiDas: Containerizing Data-Intensive Applications with I/O Specialization. In *3rd International Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS '20), June 23, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3391800.3398174

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
P-RECS '20, June 23, 2020, Stockholm, Sweden
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7977-9/20/06…$15.00
https://doi.org/10.1145/3391800.3398174

1 INTRODUCTION

Reproducible application workflows reduce the time for validation. This increases the trust in their results. However, reproducing data-intensive application workflows continues to be a challenge in the domain sciences [14]. We consider data-intensive applications that can be entirely reproduced when provided along with their code, data, and environment specification. We focus particularly on applications that access a large amount of structured data, using custom I/O-optimized libraries. Such program are common in the scientific domain, where data is stored in formats such as NetCDF4 [15], HDF5 [9], or UniProt [3]. Most of these formats are self-descriptive – that is, the data and metadata are stored in the same file. Being self-descriptive, they require specific I/O libraries for reading and writing these formats to be included with the application.

Partial evaluation [5] offers a strategy for automating the process of pruning the codebase. It is an optimization technique that uses knowledge of static inputs to generate a specialized version of a program that only accepts the remaining dynamic inputs. Figure 1 shows a piece of code in which the height of a building (Line 4) is computed by calling the `compute_opposite()` function which multiplies building distance by the tangent of the viewing angle. Since Line 10 involves a call to the *tan()* function, the entire math library *libm* must be included to compute the function. Partial evaluation determines the static `viewing_angle` value and creates a specialized version of the `compute_opposite()` function, as shown in Figure 1. Note that the *tan()* call is also replaced with 1 (on Line 7 in Figure 1), which is the result of evaluating it with argument `pi/4`. Since a call to *tan()* is no longer necessary, the math library can be eliminated from the specialized version of the program.

2 MIDAS: I/O CALL SPECIALIZATION

The LLVM [11] toolchain supports static compilation and analysis of code. Its C/C++/Objective-C source frontend, `clang`, can produce *bitcode*, an interpretable intermediate representation. The included backends can compile this to native binaries for a range of hardware architectures. LLVM provides a code optimizer, `opt`, that runs specified transformations or analyses. Optimizations are specified in terms of *passes*. Opt operates on and modifies bitcode. We use LLVM-based instrumentation to *(i)* analyse the application and identify the accessed file regions, and *(ii)* modify the application to include the accessed data chunks. We also use Wholly!'s [6]

```
#include <math.h>
float compute_building_height(float building_distance){
        float viewing_angle = pi/4;
        float building_height =
          compute_opposite(building_distance,
            viewing_angle);
        return building_height;
}
float compute_opposite(float adjacent, float angle){
        float opposite = adjacent * tan(angle);
        return opposite;
}
```

(a) Original code

```
float compute_building_height(float building_distance){
        float building_height =
          compute_opposite_specialized(building_distance);
        return building_height;
}
float compute_opposite_specialized(float adjacent){
        float opposite = adjacent * 1;
        return opposite;
}
```

(b) Specialized code

Figure 1: *Apriori* knowledge of inputs allows the evaluation of some control flow choices and variable assignments prior to runtime, potentially resulting in dead code that can be eliminated.

gllvm [7], a tool to build LLVM bitcode files using the unmodified build scripts of C or C++ source programs.

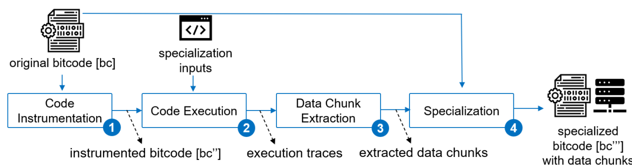

Figure 2: MiDas: Automates the identification and inclusion of data chunks required by an application

2.1 Motivating Example

Figure 3a shows an example application that reads data from a file. Depending on the high-level user input, *bytes*, the offsets of the file *test.txt* as well as the corresponding data chunks accessed vary. *Our goal is to automate the process of identifying and including the required data chunks along with this application.* To do this, we design an I/O specialization framework, **MiDas**, shown in Figure 2.

2.2 Code Instrumentation

The source code is first compiled into LLVM bitcode *bc* using LLVM compiler `clang`. This is provided as input to our LLVM transformation pass. The pass instruments I/O calls, such as *open*, *read*, and

```
void file_read(int bytes){
    int fd, sz;
    char *c = (char *) calloc(bytes, sizeof(char));
    fd = open("test.txt", O_RDWR);
    lseek(fd,100,SEEK_SET);
    sz = read(fd, c, bytes);
}
```

(a) Application source code

```
1, test.txt, 100, 150
1, test.txt, 100, 190
1, test.txt, 100, 230
1, test.txt, 100, 450
```

(b) Traces from four executions of the instrumented example application with high-level user-provided specialization input, *bytes* - 50, 90, 130, 350, respectively.

```
%94 = load i32, i32* %9, align 4
%95 = sext i32 %94 to i64
%96 = call i64 @read(i32 %92, i8* %93, i64 %95)
%97 = trunc i64 %96 to i32
store i32 %97, i32* %13, align 4
%98 = load i32, i32* %12, align 4
```

(c) LLVM assembly code segment corresponding to original bitcode: *read* call highlighted in red is the I/O call to be specialized.

```
%94 = load i32, i32* %9, align 4
%95 = sext i32 %94 to i64
%96 = bitcast [17 x i8]* @fileData to i8*
call void @llvm.memcpy.p0i8.p0i8.i64(i8* %93, i8* %96
      i64 %95, i32 1, i1 false)
%97 = alloca i64
store i64 %95, i64* %97
%loadRetVal = load i64, i64* %97
%98 = trunc i64 %loadRetVal to i32
store i32 %98, i32* %13, align 4
%99 = load i32, i32* %12, align 4
```

(d) LLVM assembly code segment corresponding to I/O-Specialized bitcode: Lines 3 to 8 highlighted in red are inserted to replace the *read* call during specialization.

Figure 3: Example application reading a file

lseek, that handle file-related operations to record the file offsets and spans accessed during execution. Specifically, we add a custom wrapper function following each I/O call. The input arguments of the I/O call are passed to the wrapper. Using these arguments, the wrapper functions construct and maintain a global data structure that associates a file descriptor or file pointer with a filename. This data structure also keeps track of the file offsets after each I/O call. Apart from the aforementioned input arguments, the LLVM pass also generates a unique identifier per I/O call site and incorporates it in the wrapper functions. The identifier is used to distinguish between multiple I/O call sites. During execution, these wrapper functions print out all the information required for specialization - I/O call site identifier, filename and file offsets. Instrumentation of the example application in Figure 3a results in the insertion of

three wrapper functions after the I/O calls *open* on line 4, *lseek* on line 5, and *read* on line 6.

2.3 Code execution with Specialization Inputs

The bitcode *bc"* generated by the above pass is then compiled into native code using clang and executed along with high-level user inputs (*i.e,* specialization inputs). Executing this instrumented native code produces execution traces as output by the I/O call wrapper functions.

Figure 3b shows traces from four different executions of the instrumented example application with high-level user input or specialization input, *bytes*, having values 50, 90, 130, 350 respectively. Each line of the execution trace contains the I/O call site identifier, filename and start, end file offsets and correspond to the *read* I/O call on line 6 in Figure 3a.

2.4 Data Chunk Extraction

Execution traces from above are provided as input to our LLVM analysis pass. Based on the file offset information from the execution traces, this pass reads the corresponding data chunks from the files. It creates a data structure consisting of these data chunks along with the associated I/O call information (*i.e,* I/O call site identifier, filename and file offsets). This data structure is passed as input to the next LLVM pass that performs specialization.

In case of the example application, the minimum and maximum file read offsets are 100 and 450. Hence the data chunk corresponding to this range is extracted.

2.5 Specialization

Bitcode *bc* and the data structure containing the file data chunks from above are passed to our LLVM transformation pass. This pass replaces the I/O calls corresponding to file read operations such as *read, fread, pread,* etc with the extracted data chunks and generates the specialized bitcode *bc"'*. In particular, this replacement involves the following operations:

(1) The extracted file data is stored in a global variable.
(2) A *memcpy* instruction is inserted to copy the required data from the global variable to the buffer argument of the file read I/O call.
(3) All variables related to the I/O call being replaced are updated to ensure correct operation of the modified bitcode. For instance, the return values of the I/O calls are updated with the number of bytes copied into the buffer argument.
(4) The I/O call instruction is removed.

LLVM assembly code segment corresponding to the original bitcode *bc* for the example application is shown in Figure 3c. I/O call to be specialized here is the *read* call highlighted in red. Figure 3d shows the LLVM assembly code segment corresponding to the specialized bitcode *bc"'*. *fileData* is a global variable here containing the extracted data chunk. Lines 3 to 8 highlighted in red are inserted to **replace** the *read* call during specialization.

3 EVALUATION

We tested our I/O specialization framework, **MiDas**, on below mentioned Python applications that access scientific data in NetCDF4 [15] format. These applications make use of NetCDF4-python module to work on the data. This module is a Python interface to the NetCDF C library. HDF5 is a high performance data software C library utilized by NetCDF for fast I/O processing and storage.

3.1 Identifying I/O calls for Specialization

To understand the behavior of NetCDF and HDF5 libraries with respect to file sizes, we first investigated on python applications accessing NetCDF4 data files of sizes - **384 bytes, 1012 bytes, 30 MB, 700 MB, 1.4 GB, 9 GB and 12.8 GB.**

In case of small files of sizes 384 bytes and 1012 bytes, high-level user inputs map to *fread* calls in NetCDF library. In particular, one *fread* call reads the metadata of size 8 bytes and another *fread* call reads the entire file - 384 bytes and 1012 bytes respectively.

In case of large files of sizes 30 MB to 12.8 GB, high-level user inputs map to *fread* call in the NetCDF library and *pread* calls in HDF5 library. Specifically, the *fread* call in NetCDF library reads the metadata (size 8 bytes) and *pread* calls in HDF5 library read subsets of file data in chunks as required.

From our experiments, we observe that *fread* calls in NetCDF library only access metadata of size 8 bytes apart from reading the entire file in case of small files. Hence, there is not much benefit in specializing this I/O call. On the other hand, *pread* calls in HDF5 library access subsets of files in case of larger files. Therefore, specializing this I/O call in HDF5 library is beneficial.

We build NetCDF C library and HDF5 C library from source to generate the LLVM bitcodes using gllvm [7]. We then modify the bitcode generated from libhdf5.so (*i.e,* the shared object corresponding to HDF5 library) by replacing the *pread* I/O call itself with data chunks required by the python application (as described in §2). Next, we compile the modified bitcode to generate a new shared object (.so) and replace the original library with this new one. Hence, whenever the python application invokes NetCDF C library through the NetCDF4-python module interface, NetCDF C library would in turn invoke the specialized version of HDF5 library.

3.2 File Access Pattern

Portion of File Accessed. In case of large data files, we identify what portion of these are accessed. For this experiment, we generated larger files of sizes upto 12.8 GB from the 30 MB NetCDF data file by rewriting the data for multiple timesteps. These NetCDF data files consist of four attributes: temperature, salinity, sea water velocities in 2 directions. These are defined as time series over the dimensions time (expressed in days), latitude and longitude (expressed in half and full degree increments respectively) and depth (expressed in meters). We consider a python application that only accesses data corresponding to the temperature attribute and not salinity, sea water velocities. When the python application accesses all the data corresponding to temperature attribute, this maps to a *pread* call in HDF5 library reading only **22%** of the data file. Table 1 shows the portion of file accessed for various file sizes. Thus, the remaining 78% of data corresponding to other attributes is irrelevant to the application. Further, *pread* calls read data in the range of tens of KB with respect to the 30 MB data file when the python application accesses only a subset of the data corresponding to

Table 1: **Portion of File accessed corresponding to 'temperature' attribute: A python application that accesses this data results in utilizing only 22% of the entire file.**

Total Size	30 MB	700 MB	1.4 GB	9 GB	12.8 GB
Accessed Size	6.6 MB	154 MB	0.3 GB	1.98 GB	2.82 GB

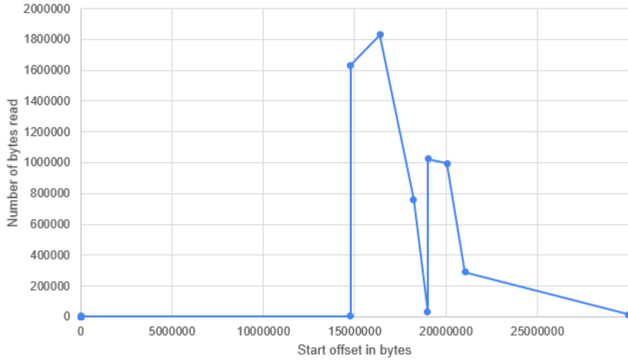

Figure 4: Location of 6.6 MB of data accessed corresponding to 'temperature' attribute in 30 MB NetCDF file: The x-axis depicts the start file offset and the y axis depicts the number of bytes read during each *pread* I/O call.

temperature attribute by applying conditions on attributes: time, depth, latitude and longitude.

Location of Accessed Data. Next, we identify the exact offset locations of the accessed data chunks in the data files. For this experiment, we consider the 6.6 MB of data accessed corresponding to the 'temperature' attribute in the 30 MB NetCDF file. Figure 4 shows the file access pattern. The x-axis depicts the start file offset and the y axis depicts the number of bytes read during each *pread* I/O call. This shows that most of the data accessed in the file lies in the offset range of 15 MB to 20 MB.

These results illustrate the fact that applications often tend to access only a subset of large NetCDF data files.

4 DISCUSSION

After specializing the HDF5 library, the python application runs correctly only when the high level user inputs map to a subset of the included data chunks. The challenge here lies in identifying the optimal file offsets based on values extracted from execution traces. Currently, MiDas completely relies on high level user inputs provided during the specialization phase. For instance, consider an application that accesses different subsets of the temperature attribute based on user inputs. Providing user inputs such that the entire temperature attribute is accessed during the specialization phase would ensure that all required data chunks are included with the specialized library.

We plan to further improve the identification of optimal file offsets by integrating invariant generation tools like Daikon[4] into our framework. An invariant is a property that holds at a certain point in a program considering all possible executions. For instance, $y = 2 * x + 3$, $5 \leq x \leq 10$ are some examples of such invariants. Daikon generates likely invariants from execution traces of an application. This could be used to identify an optimal range of file offsets from the execution traces.

Currently our work focuses on *read* I/O call and includes data chunks corresponding to this call with the application. However, specializing only the *read* calls would be unsound if the program writes to a specific file region and later reads from it. So, we plan to extend MiDas to specialize *write* I/O call and track the corresponding data chunks to update related *read* I/O calls accordingly.

5 RELATED WORK

Containers, driven by the popularity of Docker [1], have recently emerged as a lightweight alternative to hypervisor-based virtualization. Container-based deployment however is inefficient due to large size of images and redundant downloading of data within each layer [18]. Slacker [8] provides a more efficient container system based on de-duplication of file blocks. Block-level de-duplication techniques do not eliminate data redundancies within structured arrays. In this work, we have focused on reducing the size of containers encapsulating data-intensive scientific applications using I/O specialization.

Partial evaluation historically focused on functional programming languages since the absence of side-effects simplified the analysis [10]. However, operating systems kernels, performance-oriented user space libraries, and many system utilities are typically written in C/C++. C-Mix/II [12] and Tempo [2] were two early efforts that established the feasibility of specializing such code. OCCAM [13], LLPE [17], and Trimmer [16] have demonstrated that it can be applied to modern C/C++ applications by automatically building [6] them into LLVM bitcode. LLIO [17] investigated elimination of accesses to the filesystem to improve runtime performance. However, it did so by lifting entire files, an approach that is untenable for large data sets. Our strategy is similar to that of Trimmer with the difference being we focus on the input data of applications rather than their configuration files.

6 CONCLUSION

In this work we have highlighted the need for I/O specialization to efficiently reproduce data-intensive applications. Our work motivates novel support for optimizing sparse access to large files. Our I/O call specialization framework, **MiDas** identifies and includes only the relevant chunks of data along with the application in the container. This reduces the size of container and yet preserves its functionality when repeated in different environments. Our experiments show that **MiDas** is able to precisely identify the amount of effective data utilized.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant ACI-1440800 and the Office of Naval Research (ONR) under Contract N68335-17-C-0558. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

REFERENCES

[1] Docker. https://www.docker.com/, 2019.

[2] Consel, C., Lawall, J. L., and Le Meur, A.-F. A tour of tempo: A program specializer for the c language. *Science of Computer Programming 52*, 1-3 (2004).

[3] Consortium, U. Uniprot: a worldwide hub of protein knowledge. *Nucleic acids research 47*, D1 (2019), D506–D515.

[4] Daikon. https://plse.cs.washington.edu/daikon/.

[5] Futamura, Y. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation 12*, 4 (1999).

[6] Gelle, L., Saidi, H., and Gehani, A. Wholly: A build system for the modern software stack. In *23rd International Conference on Formal Methods for Industrial Critical Systems* (2018).

[7] gllvm. https://github.com/sri-csl/gllvm.

[8] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)* (2016), pp. 181–195.

[9] HDF5. https://www.neonscience.org/about-hdf5.

[10] Jones, N., Gomard, C., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[11] LLVM. https://llvm.org/.

[12] Makholm, H. Specializing c: An introduction to the principles behind c-mix, 1999.

[13] Malecha, G., Gehani, A., and Shankar, N. Automated software winnowing. In *30th ACM Symposium on Applied Computing* (2015).

[14] Meng, H., Kommineni, R., Pham, Q., Gardner, R., Malik, T., and Thain, D. An invariant framework for conducting reproducible computational science. *Journal of Computational Science 9* (2015).

[15] NetCDF. https://www.unidata.ucar.edu/software/netcdf/.

[16] Sharif, H., Abubakar, M., Gehani, A., and Zaffar, F. Trimmer: Application specialization for code debloating. In *33rd ACM/IEEE Conference on Automated Software Engineering* (2018).

[17] Smowton, C. *I/O optimisation and elimination via partial evaluation.* PhD thesis, University of Cambridge, 2014.

[18] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 1–17.