Automated Packaging of Bioinformatics Workflows for Portability and Durability Using Makeflow

Casey Robinson and Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame

ABSTRACT

Dependency management remains a major challenge for all forms of software. A program implemented in a given environment typically has many implicit dependencies on programs, libraries, and other objects present within that environment. Moving applications between different runtime environments is certain to fail due to the existence of those external dependencies.

Workflows particularly suffer from dependency management problems, precisely because they tie together multiple independent programs into a coherent whole. To address the problem of workflow decay, we propose applying the old idea of a "linker" into the new context of workflow systems. We have implemented a linker for the Makeflow workflow system, and extended the concept to apply recursively to executables and scripted languages within the workflow. We evaluate the system by applying it to a selection of bioinformatics workflows including BLAST, BWA, and SHRiMP, enabling them to be moved across multiple computation environments. We also show that the portability provided by packaging allows for improved performance.

1. INTRODUCTION

In almost every domain of computing, one can easily find a user with a program that ought to run, but cannot proceed until some mysterious program, library, or file is installed first. A program written in nearly any language cannot be moved from one computer to another without carefully understanding all of the objects upon which it depends. Despite years of practice, managing dependencies is still a usability hurdle for all forms of software.

Workflows have many characteristics which provide a particularly large susceptibility to missing or mismatched dependencies. Workflows bring together large amounts of computation and data. Workflows often combine many different programming environments and technologies. Workflows are are intended to run across thousands of distinct machines.

WORKS13 November 17, 2013, Denver, CO, USA.

ACM 978-1-4503-2502-8/13/11:.$15.00
http //dx.doi.org/10.1145/2534248.2534258

As a result, a workflow constructed in one environment has no chance of running correctly in another environment without significant effort by the user to determine all of the necessary dependencies. Zhao et al have identified this problem and given it the name of *workflow decay* [6].

To address the problem of workflow decay, we propose adapting an old idea into a new domain: workflow systems should have linkers [4]. In the most traditional sense, a linker analyzes a compiled program, determines the minimal components from a library needed to satisfy the program, and produces a self-contained executable that has all of the needed components to run. Linking workflows targets decay by capturing the namespace and files associated with each element in the workflow description. Of course, linking has tradeoffs. A self-contained package is inherently larger than an incomplete package, and may be less amenable to sharing of resources at runtime, or other forms of analysis.

We have implemented these concepts in a linker for the Makeflow workflow system [1]. Makeflow takes a list of rules written in a similar style to GNU Make, produces a directed acyclic graph of tasks to be run, and schedules them on a variety of clusters, clouds, and grids. The linker processes the dependencies found in the original execution environment and produces a packaged workflow that can be moved to other execution environments. Going beyond the traditional notion of linking, a workflow linker can be used to verify proper local dependencies, troubleshoot problematic workflows, and estimate resource consumption. While we have prototyped this idea within a specific workflow system, the ideas can be easily translated to other systems.

We have evaluated this implementation by applying it to a selection of bioinformatics workflows that have dependencies in multiple forms: such as interpreters, shared libraries, configuration files, and data. Using multiple levels of dependency processing, we show that the original workflow specifications implicitly refer to more than 100 MB of software. To evaluate the portability, we move the packages across three different computing environments, and show that processing two levels of dependencies is necessary for successful execution of the workflows. A portable package enables performance improvements since the workflow can be relocated.

Approach	Benefit	Drawback
Error Recovery	No work upfront	Separation from Workflow Creation
Execution Trace	Scientific Reproducibility	Multiple Runs
Static Analysis	Analysis before Execution	User Discipline

Table 1: Roadmap of Workflow Portability and Preservation Strategies

2. RELATED WORK

There are three general approaches to combating making a workflow portable - error recovery, execution trace, and static analysis. Table 1 outlines the costs and benefits of each approach.

The workflow can be run while monitored for errors. Each caught error is diagnosed and repaired. Eventually the workflow will run successfully. This approach requires no work upfront while allowing for successful execution. However, the catch and repair process is time consuming, complex, and unique to each error. The repair process is also separated by time from the workflow creation process. The time gap further exacerbates the repair complexity. This strategy is demonstrated by Zhao et al using Taverna as an example [6].

Observing a successful run of the target workflow and recording every action produces a log which exactly describes the workflow. The recorded trace is then saved for future runs of the workflow. From the trace we can collect each file accessed and have a workflow with no external dependencies. This approach is ideal for scientific reproducibility, as there is a log of every system call made. However, in order to collect the log the workflow must run successfully. Requiring multiple runs of a workflow provides no assistance in running across heterogeneous, grid systems. For example, the library loaded for parsing the input data may be defined by an environment variable that is different on each machine. This is the strategy implemented with CDE [2].

Static analysis of the workflow is a third approach to preservation. By looking at each piece of the workflow before execution and attempting to find its dependencies, a self-contained, transportable packaged workflow can be created. Static analysis is less time consuming than the other approaches. Workflows may have multiple execution paths with different dependencies. Static analysis requires one run while the other methods require multiple runs to capture the dependencies from each path.

There are limits to this approach. It is impossible to find every dependency without running the program. The user must interact with the linker and understand the problem of dependency management before executing the workflow. This is the approach we discuss in this paper. We chose static analysis of formal dependencies because it can be used before execution of the workflow.

3. A MODEL OF LINKING

For the purpose of linking we use the term dependency to mean *data* dependency - files that must exist in order to run an application. A dependency is any piece of software which is required in order to run an application. Dependencies come in a wide variety of types; shared libraries, other executables, scripting language interpreters, and input files. Each program has implicit and explicit dependencies which typically correspond to objects in a file system. Implicit dependencies are those dependencies which are revealed at runtime or are not defined by the user, e.g. the kernel version. Explicit dependencies are defined formally by the user via some sort of `import` statement or listing. Reliable distributed computing across heterogeneous resources is impossible unless every dependency is explicitly defined. Consequently, the focus is on explicit, or formal, dependencies for the task of linking.

Dependencies may have their own dependencies, recursively creating a dependency tree. Describing the construction and traversal of a dependency graph is a key contribution of this paper.

Name resolution is the fundamental problem tackled by linking. Dependencies are defined by a name which is defined in a localized context. When discussing linking, the name is a relative path and the context is the file system combined with the current working directory. The linker must collect the local name as well as the namespace in which that name is defined and provide a method for replicating the name lookup at the execution site. The process of linking makes the underlying name translation table explicit.

Linking is broken down into three steps: parsing, discovering, and collecting. Parsing reads the workflow description, generates a tree of dependencies, and prunes the dependencies generated during execution. Discovering is the process of recursively searching each dependency for further dependencies and storing the results in a tree. Collecting iterates over the tree and creates a self contained version of the workflow called the packaged workflow, which is the outermost package and invoked identically to the original workflow description. Figure 1 is an overview of the steps of linking and an example of the corresponding data structure.

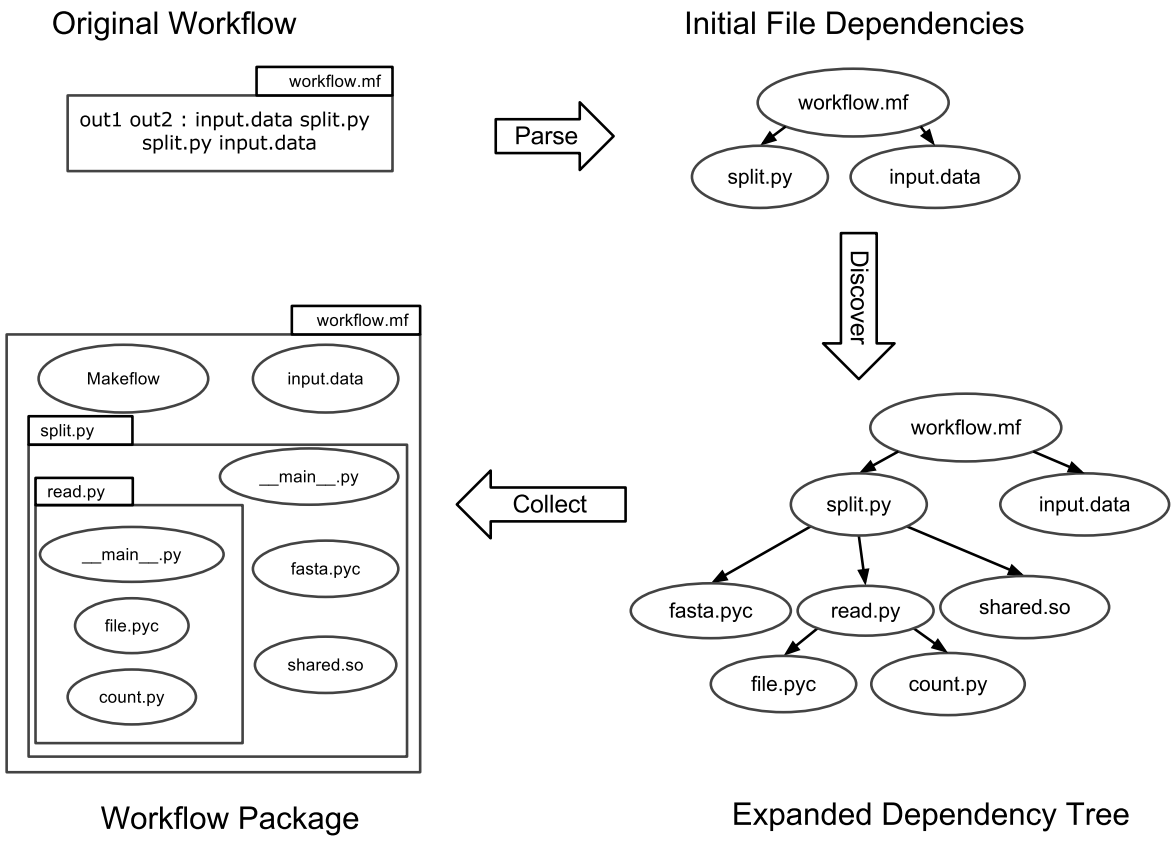


Figure 1: Steps of Linking

3.1 Package Responsibilities

A package has two responsibilities: invocation consistency and encapsulation of the dependency graph. These two responsibilities work together to emulate the namespace resolution in the original setting. Invocation consistency means that the packaged dependency is started with the same command. This can be accomplished by manipulating path variables or using magic files. The `PATH` variable stores a list of file system locations in which to look for a dependency. `PATH` manipulation consists of inserting additional locations which will contain the collected dependencies. Magic files have a special name which is known by the execution environment and contain instructions for execution. For example, the Python interpreter when given a directory looks for a file named `__main__.py` inside that directory. In order to be transportable the package must store all of the dependencies. The storage layout is arbitrary, and can vary amongst file types or even individual files, provided each file can be found.

3.2 Categories of Dependencies

A complex workflow can have many dependencies which fall into different categories. Figure 2 provides a hierarchy of the dependency categories.

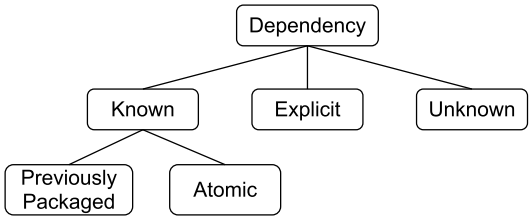


Figure 2: Dependency Type Hierarchy

This case represents files which the linker has been programed to handle. Known dependencies correspond to static linking; dependencies are found and relocated into the package upon linking. There are two key requirements for a file type to be known. The linker knows how to find the formal dependencies expressed in a file ('import/include statements', 'ldd output', etc.). The linker also knows how the target file finds its dependencies (search path, table lookup, etc.), and can therefore redirect the packaged version of the file to its relocated dependencies.

Atomic files (typically data and configuration) are a subset of known dependencies and are the simplest case for linking. An atomic file contains no external references and can therefore be safely copied into the package without further processing.

Explicit dependencies are software packages which are defined by a name and a version. Explicit dependencies are best suited to common packages - language interpreters, domain specific libraries, reference data sets - which will likely be available at all of the target execution sites. Handling explicit dependencies is analogous to dynamic linking. Thus the same drawbacks and benefits apply.

For example, a python script depends on a certain version of the Python interpreter. The version number is sufficient to identify the dependency and ensure compatibility in the future. Updating the interpreter's version does not require re-linking and the package will have a smaller disk footprint. However, the package is no longer self contained and the potential pool of systems for which the packaged workflow will run without additional effort shrinks.

Determining the version number directly follows from the assumption that the workflow is correctly configured. However, versions different from the version found on the system during linking may be suitable. In this case the user can specify a range of versions which are compatible.

Previously packaged dependencies have specific structure to satisfy the self-containment and invocation consistency requirements. Linking with previously packaged dependencies requires an understanding of the package structure and defining a method for composition. While the exact nature of composition depends on package structure, methods exist for each structure.

In the recursive scenario packages are self-contained and can be considered atomic and blindly copied. If, however, we desire to combine all of the dependencies into one directory the package will need to be reduced into its base components, each of which is added to the tree as a child of the node representing the package. Re-linking the package is potentially simpler since all of the dependencies are in one location. Name conflicts are the only potential point of contention. Linking directly solves conflicts. Thus with either recursive or single directory packages are composable

Unknown files represent dependencies for which no driver has been defined. There are two options for files in this category; copy or ignore. Ignoring the dependency leaves a hole in the package and increases the potential for failure when attempting to run on another system. Blindly copying the dependency increases the chance of success, but only slightly. With either method unknown files are detrimental to the success of linking. Consequently, the most important action for the linker to take upon encountering an unknown file is to inform the user and allow their feedback to inform the next action.

3.3 Summary

Atomic, unknown, and explicit dependencies represent the leaves of the dependency graph while known dependencies are the inner nodes. There is no clear-cut rule about how to handle all of the leaves. Some definitely should be copied (bash scripts), some should definitely not be copied (device files), and some are unclear (locale). The default is to blacklist known incompatible files and copy the rest.

4. IMPLEMENTATION

We built a linker which traverses the dependency graph of a workflow and collects every formal dependency along the way. For each formal dependency we determine the file type and pass the dependency off to the corresponding driver. We implemented drivers for common scripting languages, executables, and shared libraries. Our linker was built for Makeflow, but the concept, and file drivers, generalize to other workflow management systems.

Makeflow represents workflows as a directed acyclic graph consisting of commands connected by files. Makeflow also requires each rule to list all of the files required to run and the files which will be produced. When taken as a whole, the set of files which are inputs to rules consist of two types: files created as output of another rule and files which are inputs to the workflow. For the linking problem we are interested in the latter category, which can be found by computing the intersection of the set of input files and the set of output files. Due to the semantics of a Makeflow rule, the previously computed set of files are the only requirement to link a workflow. The next step is to link each of the inputs.

Recursively finding dependencies will collect every necessary file. However, not all of these files should be copied. Consider the `libc` library, it defines the system calls accessible to user-space programs. These functions, by definition, are system specific. Thus, proper execution of the workflow must use the `libc` library installed at the execution site.

4.1 Overview

While it is possible to start linking with any file, we chose to start with the workflow description. Linking begins with a correct workflow. A correct workflow is important because it means that the computer contains all of the information required for a successful run. This information is stored in a variety of formats in a variety of locations, but they all exist and can be accessed. From this starting point the linker can traverse the dependency tree, relocating dependencies into the package along the way.

The linking algorithm employs a queue to store pending dependencies and a list of processed dependencies. Dependencies are removed from the front of the queue and processed until the pending queue is empty or the user specified depth has been packaged.

From the Makeflow file the linker obtains a list of formal dependencies and inserts each into the pending queue. The linker then discovers the file type of each file and passes it off to the corresponding driver. The driver determines the formal dependencies of the file and returns a list to the linker. The linker then inserts each dependency into the pending queue. This process continues until every node in the dependency tree has been processed.

As dependencies are inserted into the pending queue they are annotated with information about their position in the dependency graph and their file properties. The bookkeeping includes: the distance from the root, the parent, and the ancestor which is a formal dependency of the workflow. The file properties include determined file type (to avoid recomputation and to inform descendant file type selection)

and absolute path on disk. This information allows precise control over the graph traversal and final package structure.

4.2 Discover File Type

Not to be confused with dependency categories discussed in section 3.2, a file's *type* is an identifier which corresponds to the driver that will be used for the file. Accurately determining the file type is difficult [3], so the linker looks for a few key pieces of information and if none exist the file is deemed "unknown". The most obvious indicator for scripts is the shebang line, usually located at the top of the script. The shebang not only indicates which language but gives a direct path to the executable the user desires to use for interpretation. This is the only formal dependency used for determining file type. The shebang is also used to find the desired interpreter and its version number to list as an explicit dependency. The file extension is a strong, but informal, indicator of which programming language and execution environment is desired. The last resort is the UNIX utility `file`. `file` probes the first few bits of a file and matches them against a database, `magic`. The first two tests usually match data and source code while `file` matches libraries and executables.

4.3 File Types

The main goal of the linker is to wrap each dependency in a self-contained, executable archive. This requires an understanding of how each type of file locates it's external dependencies. This subsection discusses the methods by which each file type handles external dependencies and how the lookup behavior is modified to create an encapsulated package.

4.3.1 Makeflow

Makeflow represents a workflow as a series of commands connected by common files. Each command lists required input files and output files produced. From these lists the linker distinguishes between intermediate files and the inputs to the workflow. The later category represents the list of targets which the linker needs to collect. Makeflow provides a list of these workflow inputs which are then added to the linking queue.

4.3.2 Standard Executable

Standard executables are handled by Starch [5]. To ensure correct execution Starch creates a self extracting archive which includes other executables and libraries required by the executable. Executables are found by traversing the system `PATH` variable. Shared libraries are found with `ldd` or `otool`, depending on operating system. Once all of the dependencies are found Starch copies each file into the archive and creates the setup script. A self extracting archive contains all of the necessary dependencies required to run. Execution of the archive is managed by a shell script which prepends the current directory to PATH and LD_PATH before running the original executable.

4.3.3 Python

The primitive of namespace resolution in Python is "modules" and expressed via the `import` statement. The linker searches through a file for these statements and extracts the module name. Python provides access to the internals of the `import` statement with the `imp` module. The most important function provided is `find_module` which looks for a module name in locations on the file system provided by the search path (`sys.path`) and returns the absolute path to the module if it exists. The linker takes this path and copies the file into the package. Python conveniently prepends the current directory to the search path when executing a script. The linker takes advantage of this behavior by dumping the dependencies into the directory containing the target script and consequently does not require any modifications to run at the execution site.

4.3.4 Perl

As with Python, Perl programs express their dependencies with a special statement containing either `use` or `require`. The Perl interpreter locates dependencies by searching through a list of paths stored in the `INC` variable. Unlike Python, Perl appends the current directory to the search path. This slight change in behavior can have dramatic consequences for linking. To avoid conflicts the linker creates a wrapper script which prepends the current directory to the Perl search path before loading the target script.

4.3.5 Explicit dependencies

Upon completion the linker produces a list of the explicit dependencies which are found by combining `which` with the shebang line from top level scripts or file extensions. This list is the executables and libraries which need to be available at an execution site to ensure successful execution of the packaged workflow. It is possible to communicate these requirements with batch systems and only schedule tasks on compatible machines.

4.3.6 Completeness

Our implementation includes drivers for the file types we encountered in the example workflows, but many other file types exist and are categorized as "unknown"". Fortunately, the modular structure of the Makeflow linker allows for simple extension.

Three steps are required to add support for a new file type. A method of discovery must be defined which does not collide with previously defined types. The file extension method is the most straight forward. The second step is driver creation. The driver is responsible for locating dependencies given an input file. The final step is choice of packaging method which allows the package to be self-contained and maintain invocation consistency. The choices are recursive and coalesced which we now discuss.

4.4 Package Construction

At this stage of linking, the complete queue contains enough information to construct the package. There are two distinct approaches to construct the package: *recursive* or *coalesced*.

The *recursive* method is based on a bottom-up traversal of the dependency tree. At each node the linker would create a new package for the node. At the leaves a recursively constructed package would simply contain the dependency itself. For the next level up, the package would contain the dependency as well as its direct descendants nested inside. This process would continue until the root of the tree is

reached. The final structure would closely resemble the tree discovered during linking, see figure 1.

The *coalesced* method is a top-down approach. In this approach the goal is to create a single namespace containing all of the required dependencies. Top-down is not strictly necessary, but it aligns closely with our goal of maintaining names similar to those given by the user in the original workflow description. The user is acutely aware of the files at the top of the tree, while the dependencies towards the bottom may reflect obscure UNIX details orthogonal to the workflow's goal. The challenge here is to avoid naming collisions while maintaining the multiple lookup methods required by the potentially diverse set of tools used in a scientific workflow. The main weapon for constructing a coalesced package is the name lookup table. The name lookup table is a list of key-value pairs where the key is the name used by other objects in the tree to find the dependency and the value is the absolute file path. Starting at the top of the tree (or the front of the complete queue), each file is added to the table. Dependencies are continually added until the pending queue is empty.

Naming conflicts become an issue in either method. Resolving a naming conflict boils down to modifying the name and ensuring that references from other objects in the tree are correct. There are two methods for name modification: direct manipulation and encapsulation. Direct manipulation is straightforward; append a random string until the names differ and update the references accordingly. Encapsulation is handled by prepending a namespace to the file (typically a directory).

The approach we used combines both encapsulation and manipulation. Direct manipulation is used for Makeflow and Perl naming conflicts while encapsulation for all others. Direct manipulation is static, and therefore preferable. However, direct manipulation requires an intimate knowledge of name lookup in the target system. Name lookup becomes evident during runtime which may require modification of environment variables or additional parameters to interpreters. But, direct manipulation of names is supported by all target file types. And since the structure of other file types is externally defined, encapsulation is a more durable approach for most file types.

4.5 Composability

Composability of packages must also be addressed. Both approaches are composable, albeit with dramatically different costs. Recursively structured packages are trivially composable; a blind copy will suffice. Coalesced packages require merging the lookup tables and resolving conflicts. While there are cases where concatenating the tables will suffice, in the general case the entire linking process will need to be rerun.

One final consideration of package construction is invocation consistency. Invoking a package may differ from the method for invoking the original file. This can be solved by updating every reference to the package, or by modifying the structure of the package. The linker employs augmentation of the package structure to allow consistent invocation from an outside observer, Makeflow in this case. This requires modifying paths, changing the current directory, and setting environment variables. The end result is packages which may be invoked with an identical command to the original.

5. THE LINKER USER INTERFACE

The goal of the linker is to fill in the gap between the user's knowledge of the workflow description and the implicit configuration hidden in the system's various interpreters. Since the linker explicitly shows the dependencies in a workflow, the linker will also be useful during workflow construction. This will be especially useful in scenarios where multiple scientists are collaborating on a single workflow or multiple integrated workflows.

```
$ makeflow -b bwa_workflow.mf packaged_bwa
  Packaging bwa_workflow.mf into packaged_bwa...
  Skipping libc
  COMPLETE

  Package Size
    6.47 GB

  Explicit Dependencies
    Perl 5.12
    libc 2.12
    bash 4.1.2
```

Figure 3: Example User Interface

One important question to ask is "What information should be visible to the user during the linking process?". The answer depends on the file type. For atomic and known files little or no information is required. Explicit files will be listed in a separate file, but it would be useful to mention that explicit dependencies were used and point the user to the file containing the list. Unknown files represent potential failures which may require user intervention. Therefore it is vital to clearly state the status of any and all unknown files encountered during linking. Finally, a summary of files copied, renamed, and skipped should be produced. This information may be presented in graphical form or as a series of lists, one for each Makeflow dependency. Presenting the found dependencies provides two benefits: inform the user of potential naming conflicts, allow the user to understand which versions of programs will be collected.

The main concern of users during linking is determining when to stop recursing. The default is to collect everything, but the user should be able to stop linking earlier. The most basic way is to explicitly define the version of an interpreter to use. For example, use Python 2.7.

The linker should also allow the user to define which name points to which executable. If the user knows that Python 2.7 is available at all execution sites is it necessary to collect all files included in the standard Python installation? While using an explicit dependency, "use Python 2.7", will result in a smaller package and remain compatible with the current infrastructure, the workflow will decay and require manual installation of Python 2.7 in the future. This comes back

to the purpose of linking. Some users intend to create scientifically reproducible applications/experiments while others want to use disparate resources not under their control. Users looking for the first case should investigate CDE [2].

6. EVALUATION

To evaluate the linker, we selected workflows generated by an active bioinformatics web portal. The selected workflows are available for download at `http://www3.nd.edu/~ccl/workflows/`. Currently, three types of computations (BLAST, BWA, and SHRiMP) serve as the test cases for the Makeflow linker. Each workflow has a similar structure: split the input file, run computation on each split against a reference data set, and combine the results.

An important attribute of a packaged workflow is the file size. Figure 4c shows the dependency tree for an example SHRiMP workflow. The figure also annotates the tree with file size per level. The total size of the packaged SHRiMP workflow is 708.65 MB. Data sets dominate the package size, accounting for 99.6% of the total file size. This result indicates that the additional file transfer required for linking is negligible.

By varying the depth to which we traverse the dependency graph we can change the degree of linking. A shallow traversal will result in a smaller package but has less portability compared with a deep traversal. The goal of this section is to determine the minimal amount of linking required to run a workflow. Table 2 summarizes the depth analysis and indicates the required depth with a bold typeface.

Depth	BLAST	BWA	SHRiMP
0	3.5 KB	191 KB	932 KB
1	**+8.4 GB**	**+6.47 GB**	+677 MB
2	+47.2 KB	+2.88 MB	**+236 B**
3	+3.6 KB	-	+524 KB
Total Size	8.4 GB	6.47 GB	678 MB

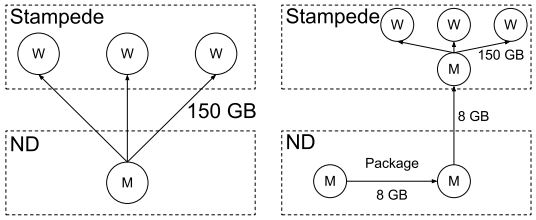
Table 2: Depth Required for Success

To evaluate the necessary level of linking for a successful run, we ran the workflows on various grid platforms around the country - CRC[1], Future Grid[2], Lonestar[3], and Stampede[4]. Level 1, the workflow description and its immediate dependencies, was the minimal level required for success with BLAST and BWA. The SHRiMP workflow required linking to level 2 - the utilities file `utils.py`.

This result was unexpected, but logical since the clusters are set up for running scientific applications and most of the Perl and Python dependencies are included with a standard installation.

6.1 Data Transfer

The increased portability provided by packaged workflows creates opportunities for improved performance. For example, packaging workflows can reduce total execution time by localizing the data transfer. To demonstrate this effect we ran the BLAST workflow in two configurations. The BLAST workflow and all of the corresponding dependencies are available at The University of Notre Dame (ND), but we want to run our workers at The Texas Advanced Computing Center, specifically the Stampede cluster. The network speed between ND and Stampede is a bottleneck in the execution of this workflow.



(d) Without Packaging (e) With Packaging

Figure 4: Executing Blast Workflow with Stampede

In the first configuration, figure 4d, we run the master process at Notre Dame and the workers as batch jobs at Stampede. As part of task distribution, the master process sends each worker all of the necessary files. Among these files is an 8 GB reference database which is common to all tasks. The master must send this file to each worker over the WAN between ND and Stampede.

In the second configuration, figure 4e, we run the master process and the workers at Stampede. To enable execution at Stampede we use the linker to package the workflow at Notre Dame and `scp` the package to Stampede. In this scenario there is only one transfer between ND and Stampede.

File transfer time from Notre Dame to Stampede without packaging totals 1,535 seconds. Package creation takes 8.4 seconds, transferring the package requires 81 seconds, and transfer inside Stampede requires 3 seconds per worker or 57 seconds with 19 workers. After accounting for the additional cost of package creation, linking provides 1,397 seconds less total transfer time - an 89.6% savings.

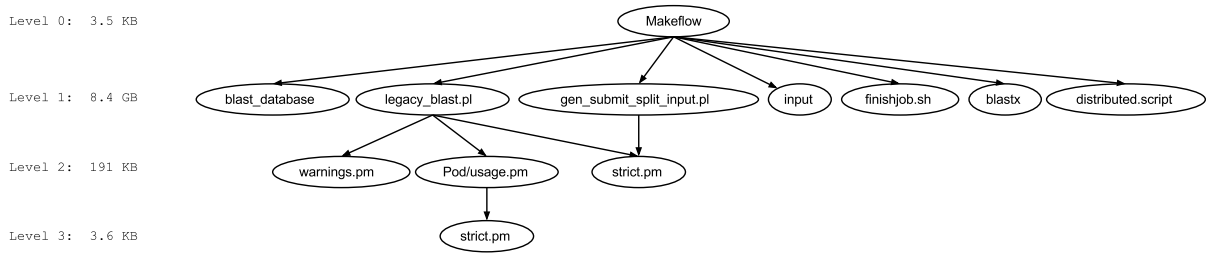
This result would not be possible without linking. The package creation time would be significantly longer and require user intervention.

7. LIMITATIONS

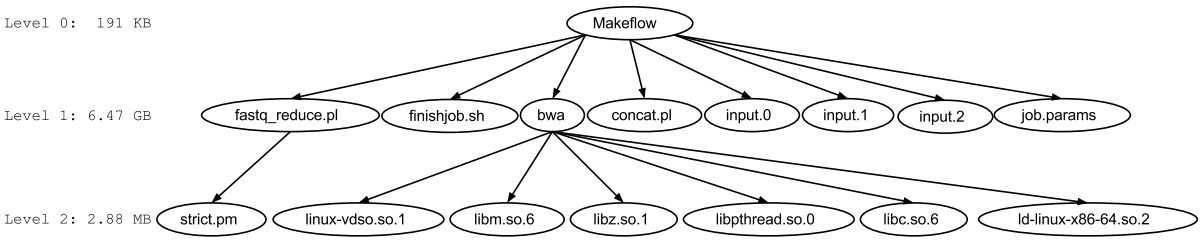
The linking approach has inherent limitations. Guaranteeing that all required files are found is impossible. For example, consider a program which employs `eval` to run code based on the type of input file. Statically analyzing such a program cannot possibly find the correct file for all inputs since the decision is made at runtime.

The linking approach also requires a method for redirecting queries to packaged objects instead of local files. Modifying the search path and rewriting every reference are the two methods by which redirection may be enabled. Rewriting every reference is untenable, which implies that an execution environment must support path modification in order to be amenable to linking.

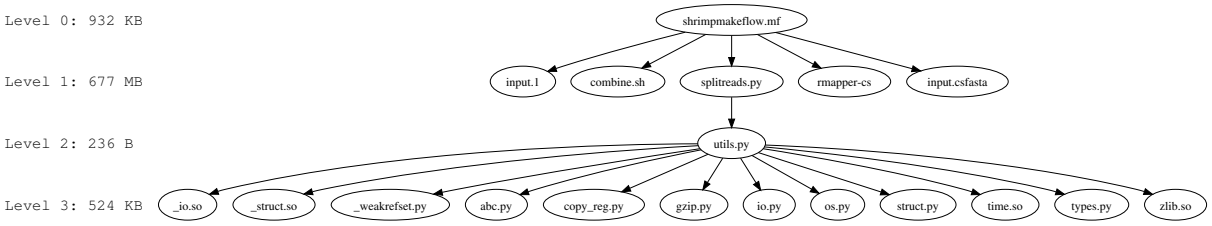
[1] `http://crc.nd.edu/`
[2] `http://futuregrid.org/`
[3] `http://www.tacc.utexas.edu/resources/hpc/lonestar`
[4] `http://www.tacc.utexas.edu/resources/hpc/stampede`



(a) BLAST Workflow Dependencies



(b) BWA Workflow Dependencies



(c) SHRiMP Workflow Dependencies

8. FUTURE WORK

In order to further evaluate the effectiveness of linking a larger and more diverse selection of workflows should be collected. The effectiveness of linking with regard to decay prevention also requires exploration, but cannot be evaluated until time has passed and systems have changed. Reducing the number of files which fall into the unknown category will increase the number of applications where linking is a viable solution. Combining linking with other forms of decay prevention may lead to increased application coverage or decreased file size and potentially increase throughput.

9. CONCLUSION

Workflows suffer from dependency management issues due to their diverse execution environments and agglomeration of applications. To address this issue we applied the concept of "linking" to workflows. Linking workflows fill the gap between the user's view of the system and the details required for execution without requiring the user to supply an exhaustive listing dependencies. Collecting the dependencies for bioinformatics workflows enabled successful execution across multiple grid systems which previously required haphazard installation of many applications.

10. ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation grant OCI 1148330 and Department of Energy grant 421K072 via subcontract from the University of Wisconsin.

11. REFERENCES

[1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[2] P. J. Guo and D. Engler. Cde: Using system call interposition to automatically create portable software packages.

[3] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 64–71. IEEE, 2005.

[4] L. Presser and J. R. White. Linkers and loaders. *ACM Computing Surveys (CSUR)*, 4(3):149–167, 1972.

[5] A. Thrasher, R. Carmichael, P. Bui, L. Yu, D. Thain, and S. Emrich. Taming Complex Bioinformatics Workflows with Weaver, Makeflow, and Starch. In *Workshop on Workflows in Support of Large Scale Science*, pages 1–6, 2010.

[6] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble. Why workflows break: Understanding and combating decay in taverna workflows. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–9. IEEE, 2012.