A COMPILER TOOLCHAIN FOR DISTRIBUTED DATA INTENSIVE SCIENTIFIC WORKFLOWS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Peter Bui

Dr. Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
June 2012

A COMPILER TOOLCHAIN FOR DISTRIBUTED DATA INTENSIVE SCIENTIFIC WORKFLOWS

Abstract

by

Peter Bui

With the growing amount of computational resources available to researchers today and the explosion of scientific data in modern research, it is imperative that scientists be able to construct data processing applications that harness these vast computing systems. To address this need, I propose applying concepts from traditional compilers, linkers, and profilers to the construction of distributed workflows and evaluate this approach by implementing a compiler toolchain that allows users to compose scientific workflows in a high-level programming language.

In this dissertation, I describe the execution and programming model of this compiler toolchain. Next, I examine four compiler optimizations and evaluate their effectiveness at improving the performance of various distributed workflows. Afterwards, I present a set of linking utilities for packaging workflows and a group of profiling tools for analyzing and debugging workflows. Finally, I discuss modifications made to the run-time system to support features such as enhanced provenance information and garbage collection. Altogether, these components form a compiler toolchain that demonstrates the effectiveness of applying traditional compiler techniques to the challenges of constructing distributed data intensive scientific workflows.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

First, I would like to thank my family for their encouragement throughout my time in graduate school. I especially owe a debt of gratitude to my wife, Jennifer Rising, for her infinite patience and unwavering support.

I would also like to thank my friends, colleagues, and collaborators who not only provided encouragement, technical help, and moral support, but also intellectual stimulation and inspiration. In particular, I thank my colleagues Michael Albrecht, Hoang Bui, Patrick Donnelly, Dinesh Rajan, and Li Yu, my collaborators Badi Abdul-Wahid, Rory Carmichael, Irena Lanc, and Andrew Thrasher, and my undergraduate advisees, Kevin Partington and Samuel Lopes.

I am also grateful for the guidance and mentoring of my advisor, Dr. Douglas Thain, whose insight and intellect routinely challenged me to improve and expand my research and myself.

Finally, I would like to thank my committee members, Dr. Scott Emrich, Dr. Patrick Flynn, and Dr. Jesus Izaguirre, who were supportive of my research and allowed me to collaborate with their students to produce the work in this dissertation.

CHAPTER 1

INTRODUCTION

Today, research scientists face the challenge of generating, processing, and analyzing a deluge of scientific data in a timely and organized manner [53]. Fortunately, they have access to an abundant amount of computing resources available to them in the form of distributed campus clusters, parallel computing grids, and commercial cloud environments. One of the key challenges that these researchers face is how to effectively and efficiently harness the computing power of these distributed systems to accelerate and scale their data intensive scientific workflows.

Fulfilling the role of the toolsmith [16], computer scientists have begun to address this problem by developing new distributed programming software tools that simplify and ease the use of such computing resources in processing this scientific data. Some of these new programming tools come in the form of distributed computing abstractions such as MapReduce [35] and All-Pairs [78] that optimize specific patterns or models of computation. These new systems generally come from the nascent cloud computing field and have been proven useful in enabling the development of high performance and high throughput distributed scientific applications [41, 110].

Unfortunately, while these distributed computing abstractions have been successful in facilitating specific patterns of computation, they often fail to encompass

large and sophisticated scientific workflows. This is because while many computational data processing workflows consist of a series of separate computational stages, these tools normally focus on a single particular stage. Such multi-stage workflows are often too complicated to be performed in a single abstraction and may in fact require the use of multiple computational abstractions. Therefore, while computing abstractions are powerful and easy-to-use, they are not general or flexible enough to support all scientific workflow patterns.

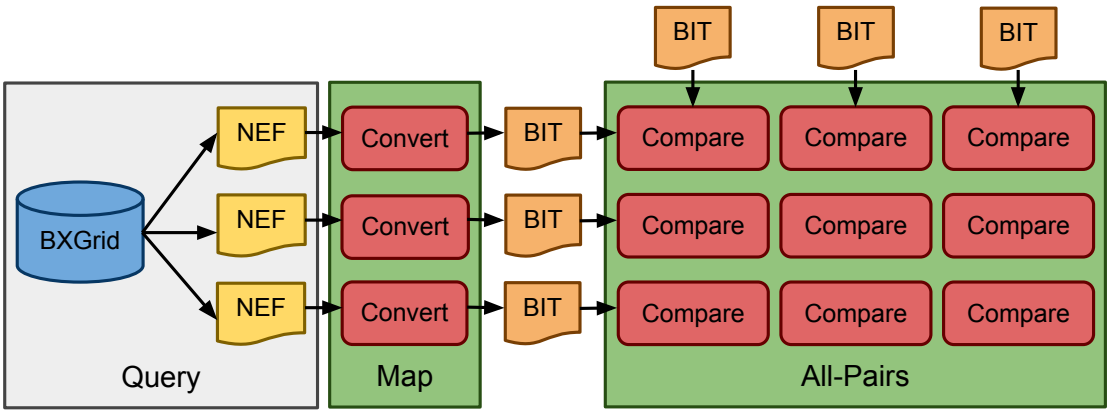


Figure 1.1. Typical Biometrics Workflow

A typical biometrics experiment involves extracting data from a repository, transforming it into an intermediate format, and then analyzing it using domain-specific tools.

For instance, researchers in the Computer Vision Research Laboratory (CVRL) at Notre Dame typically perform a experimental workflow that consists of **(1)** selecting and extracting a subset of data from a repository, **(2)** transforming the raw

data into an intermediate form suitable for processing, **(3)** and finally performing the experiment as shown in Figure 1.1. The first step involves defining a collection of input data and filtering it into the subset required for the experiement. The second step requires converting all of the elements of this selected dataset from the original raw format into a distilled version suitable for comparison and analysis by domain-specific applications. The final step takes all of the transformed data and computes a scoring matrix by comparing every pair-wise combination of the input dataset. Because each of these three steps is a unique pattern of computation, it is necessary for a workflow that encompasses all of these steps to support multiple abstractions that are capable of implementing these patterns.

In the more traditional cluster and grid computing world, the problem of multi-stage workflows has been address by distributed systems such as DAGMan [1], Pegasus [36], Kepler [75], and Taverna [86], which allow users to specify a pipeline of computational tasks. These specifications usually consist of relationships between tasks and the data inputs and outputs and are used by the software tools to construct a directed acyclic graph (DAG) representing the flow of data through the pipeline. Distributed computing abstractions can be incorporated into these systems by implementing the abstraction directly as nodes in the DAG or by using a specialized implementation as a single node in the graph [25, 60, 89, 117, 125]. Once a DAG has been formed, it is processed by a workflow manager which dispatches tasks to a distributed computing engine such as Condor [113], SGE [48], or Hadoop [51].

The main disadvantage of these DAG-based systems is that they require end users to *explicitly* construct workflow graphs, which is often cumbersome and tedious for larger sophisticated workflows [21, 23, 29, 59]. For example, it is not

uncommon for scientific workflows to consist of thousands to millions of tasks, where each job must be specified as a DAG node. Recent research projects such as Swift [120, 126] tackle the problem of efficiently specifying scientific workflows by proposing new programming languages. In these systems, the directed acyclic graph is *implicitly* constructed by a compiler or interpreter that parses and processes a workflow specification written in a high-level scripting language. The advantage of this approach is that it allows for rapid construction of sophisticated distributed applications in *concise* and *maintainable* workflow applications [119].

The introduction of a new programming language, however, has significant challenges in terms of adoption and ease of use. For instance, it is not always possible to deploy the new system onto unprivileged distributed resources, and it may be difficult to convince non-expert users to adopt the new language due to the unfamiliar syntax or programming model. Rather than developing a new language, my dissertation addresses the problem of enabling both novices and experts to develop distributed data intensive scientific workflows by proposing a workflow compiler named **Weaver** [21, 23] that is built on top of an existing general purpose programming language, Python [96]. By building the compiler as a domain-specific language (DSL) in Python, Weaver enables researchers to effectively and efficiently script scientific distributed workflows using a familiar and ubiquitous programming language with a rich ecosystem of existing software, documentation, and community.

As noted previously, the problem of enabling both expert and novice users to effectively utilize the abundant computing resources available to them is an active and ongoing topic in distributed systems. While robust and scalable systems exist for managing resources and scheduling tasks, it is still difficult, particularly for

non-specialists, to effectively harness distributed systems such as campus grids or cloud data centers. Although software tools such as abstractions and workflow systems exist, they are either not flexible enough for large scale distributed applications or too cumbersome to utilize effectively. What is needed is a way to integrate both of these types of systems.

This dissertation asserts that we need a **compiler toolchain** that enables scientific researchers to take advantage of the power of distributed computing abstractions along with the flexibility of DAG-based workflow systems. To accomplish this goal, my dissertation combines ideas from cluster, grid, and cloud computing with conventional compiler and programming language techniques into a workflow compiler toolchain facilitates the construction of distributed data intensive scientific workflows.

By applying concepts from traditional compilers, linkers, and debuggers to the challenges of distributed workflows, we will be able to improve the expression, performance, portability, and management of the resulting workflows.

With this in mind, I approach the problem of composing distributed workflows by drawing parallels between the programming and execution model of traditional programming languages with the proposed workflow toolchain. As illustrated in Figure 1.2, there are striking similarities between conventional compiler and the workflow toolchain. Conceptually, the programming and execution model found in both systems involve: **(1)** a compiler that translates a high-level representation of the application into a low-level object format and **(2)** a virtual machine that executes the generated objects by dispatching work to a target execution platform.

For instance, in a traditional programming language such as Java [49], we start with a program specified in high-level source code and pass it to the Java compiler to translate the program specification into low-level object code (i.e. class files). These generated objects can then be passed to the virtual machine, in this case the JVM [73], which executes the program by issuing instructions to a conventional computer hardware system.

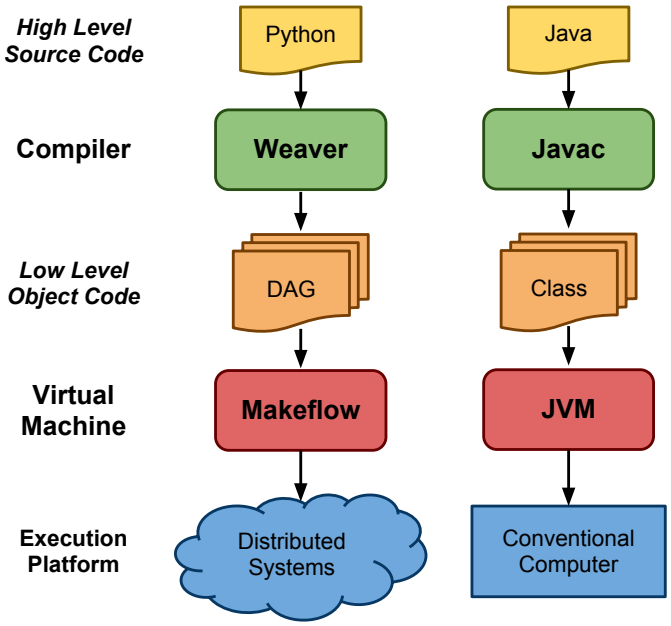


Figure 1.2. Workflow Toolchain versus Conventional Compiler.

The key insight this dissertation contributes is that one can approach the challenge of constructing distributed workflows by utilizing the programming and execution model used in conventional programming languages: translate a high-level specification of the workflow using a compiler to produce a low-level object that is then passed to a virtual machine for execution on available computing resources.

As with the traditional compiler, the toolchain proposed in this dissertation begins with a high-level specification of the application. In this case, a Python script that represents the workflow is passed to the Weaver compiler, which in turn generates a low-level DAG representation of the workflow. This DAG is then sent to a workflow manager, Makeflow [3] in this case, that executes the application by dispatching tasks to available distributed systems. In this programming model, DAGs are viewed as the assembly language of distributed workflows and abstractions are high-level functions to be combined to form these workflows.

TABLE 1.1

COMPILER TOOLCHAIN CONTRIBUTIONS

Component	Description of Research Contribution
Compiler	Apply optimization techniques to DAG-based workflow construction.
Utilities	Provide tools to package, analyze, monitor, and profile workflow.
Run-time	Modify run-time system to support local variables, nested workflows, enhanced provenance information, and garbage collection.

This dissertation applies the novel conceptual approach of compiling distributed workflows to the problem of constructing distributed data intensive scientific applications. It begins by exploring how effective various compiler techniques are

when constructing distributed workflow applications. Specifically, I investigate the application of the concepts of register allocation, instruction selection, data partitioning, software pipelining, and loop unrolling in compiling distributed workflows. Beyond these compiler optimizations, my dissertation also investigates the utility of various toolchain components such as linkers and profilers in modifying and monitoring the generated workflow objects. Additionally, I also perform some modifications to the run-time system to address some shortcomings in the virtual machine. A summary of the components of the compiler toolchain and my research contributions is provided in Table 1.1.

While my concrete contribution is the development of a compiler toolchain for distributed workflows, the dissertation also includes the construction of a few distributed applications. Some of these workflows will be synthetic benchmarks designed to demonstrate and evaluate the characteristics and features of both the toolchain and the distributed systems utilized in this dissertation. Other results, however, are from productive-level applications used by researchers from various disciplines and backgrounds. These are real world applications from fields such as biometrics and bioinformatics that benefit directly from the advances made by the research presented in this thesis.

The key impact of this work is enabling both expert and novices to effectively construct scalable data intensive scientific workflows that execute on various distributed systems. Such an ability not only increases the number of experiments researchers can perform, but also facilitates the completion of otherwise infeasible applications. Even though the dissertation focuses primarily on the design and implementation of the compiler toolchain, it is important to note that the software is already being utilized by various third parties to develop distributed

data intensive scientific workflows in the fields of biometrics, bioinformatics, and molecular dynamics, and thus is having an immediate impact on research and science beyond distributed systems.

The remainder of this dissertation describes, examines, and evaluates the various components of the proposed distributed workflow compiler toolchain and proceeds as follows:

Chapter 2 provides a review of previous research related to programming distributed data intensive scientific workflows. In particular, it first examines the use of abstractions, workflow systems, and programming languages in constructing distributed scientific applications. While abstractions are powerful in that they facilitate distributed computing without requiring the user to know the intimate details of the distributed system, they come at the cost of execution generality and flexibility. On the other hand, workflow systems enable a variety of workflow patterns but are cumbersome or difficult for novice users to utilize effectively. High-level programming language approaches provide an alternative approach that combines the ease-of-use of abstractions with the power of the more general workflow systems. In addition to discussing previous distributed computing work, Chapter 2 also describes relevant compilers and programming languages research that are incorporated into the programming toolchain proposed in this dissertation.

Next, chapter 3 discusses the Weaver compiler's programming interface and execution model. Unlike programming languages such as Swift, Weaver is implemented as a domain-specific language (DSL) on top of Python, which allows users to utilize a familiar scripting language to compose scientific workflows. Additionally, rather than introducing another workflow system, Weaver targets and

augments the Makeflow [3] workflow manager as its run-time system. Because of this, I concentrate on the programming language and compiler aspects of the toolchain, while leaving the details of distributed execution and management to Makeflow. A more in-depth examination of the programming API and implementation of the compiler is presented in Appendix A and Appendix B.

To evaluate the workflows generated by the compiler toolchain, I executed the various benchmarks and tests presented in this dissertation on the different grids, computing clusters, and cloud platforms available at the University of Notre Dame. These distributed systems are representative of the type of computing resources available to researchers at most medium to large research institutions [111].

Chapter 4 presents the application of various compiler techniques to the generation of distributed workflow applications. In particular, register allocation, instruction selection, data partitioning, software pipelining and loop unrolling are utilized by Weaver to transform directed acyclic graphs (DAG) into optimized workflows. The design and implementation of each method is discussed and evaluated using synthetic benchmarks consisting of appropriate applications and tests. The goal here is to identify how effective such traditional compiler and programming language techniques are in increasing the performance of distributed workflows and to determine when an end user should employ such optimizations to their scientific applications.

In Chapters 5 and 6, I discuss two sets of utilities that operate on the generated workflow objects. The first set is a pair of *linkers*: an application linker that packages individual commands for reliable deployment and a workflow linker that modifies the workflow DAG for portability. These linkers help users manage their software components, which is non-trivial when operating in a distributed envi-

ronment. The second set of applications consist of an *analyzer* that will examine the Makeflow log and return information regarding the execution of the workflow in a variety of formats, a *monitor* that tracks the progress of a running workflow in real-time, and a *reporter* that provides a statistical summary of a workflow's execution. These three *profiling* utilities facilitate debugging and tracking of the users workflows which are often challenging tasks for distributed applications.

In addition to developing new software tools such as a compiler and utilities, I also modify and augment Makeflow, the target *run-time* system used in this dissertation. Chapter 7 details the type of modifications made in my research work. For instance, the Makeflow parser is modified to support local task variables, which is vital for enabling the setting of task-specific batch options. Makeflow is also augmented to be made aware of nested Makeflow instances so that it can accurately perform resource allocations on hierarchical workflows. Likewise, the workflow manager is enhanced to emit additional provenance information and to support symbolic annotations. Lastly, methods for collecting garbage (i.e. temporary intermediate files) are implemented, benchmarked, and analyzed are implemented in Makeflow to prevent filesystem resource exhaustion.

Finally, in Chapter 8, I reflect on the results of my dissertation and consider possible future enhancements. For instance, I briefly explore the idea of data-mining provenance information in order automatically optimize a workflow and consider how to support dynamic workflows that allow for conditionals and run-time decision making. Additionally, I reflect on the advantages and disadvantages between having a workflow interpreter and a workflow compiler. Moreover, I discuss how the core ideas in my dissertation can be applied beyond the field of distributed computing and to programming in general.

The last decade has seen a surge in the amount of data involved in scientific research. Due to the increasing demand and requirement generating, processing, and analyzing massive quantities of experimental data, the role of distributed computing in scientific research has grown in importance across all disciplines. As such, the problem of effectively constructing distributed data intensive scientific workflows is an important, though difficult problem. This dissertation proposes a bold novel approach to constructing these data intensive workflows: a distributed workflow compiler toolchain that enables both novice and expert users to specify their workflow in a concise and maintainable high-level language which is translated to a DAG to be executed on a variety of distributed systems such as campus grids, parallel computing clusters, and cloud platforms.

CHAPTER 2

RELATED WORK

The challenge of enabling both expert and novice users to effectively harness abundant computational resources in data intensive scientific research is an active and ongoing research problem in the field of distributed computing. In particular, there has been a plethora of research into various workflow systems, distributed computing abstractions, and programming languages aimed at providing intuitive and powerful systems for constructing distributed applications. This chapter discusses the relevant work related to the proposed compiler toolchain.

2.1 Workflow Systems

Traditionally, the main approach to programming distributed applications is to provide researchers workflow systems based on the concept of directed acyclic graphs (DAGs) [123]. In these systems, the nodes of the graph are the set of tasks to be executed and the links between the nodes determine the order in each tasks are executed. Condor DAGMan [2] and Pegasus [36] are two examples of modern workflow systems that allow the user to specify a set of tasks to compute and the relationship between each task. Makeflow [3, 124] is another example a workflow manager that utilizes a DAG to schedule and manage tasks. Each of these systems provide a custom workflow language and an interpreter that takes

the job specification and produces an execution plan that is utilized by the system to process the workflow.

Another modern workflow system is Kepler [75], which is sophisticated scientific workflow application that allows users to construct workflows using a graphical interface. This system comes with many built-in components and has been used for large scale experiments. Taverna [86] is a similar graphical workflow system which builds on top of various web services and grid systems. Internally, it operates on a high-level XML language called Scufl [85], which while lacking in explicity looping constructs allows for nesting to create sophisticated data flows.

BPEL [74] is an Oasis standard for composing a set of interacting Web services into larger composite Web-based workflows [42]. Since most scientific workflows do not currently utilize Web services, BPEL's adoption in scientific data processing has been limited. Moreover, the specification itself is control-flow oriented and involves explicitly defined XML and WSDL variables and does not support dataset iteration. As such BPEL is cumbersome for computational scientists to write and often results in large and repetitive documents [118].

Though proven to be powerful and robust, these workflow systems remain underutilized due to their complexity and unfriendly user interfaces [9, 101]. That is although these workflow tools are effective and scale well to large distributed workflows, the manual construction of DAGs can be tedious and error-prone for the end user. This ineffective programming interface serves as an unfortunate barrier for many users, both expert and novices, and thus prevents such systems from reaching wider adoption and limiting their possible impact on scientific research.

2.2 Distributed Computing Abstractions

In recent years, distributed computing abstractions have been introduced to simplify the use of distributed computing systems. The most well-known such abstraction is Google's Map-Reduce [35]. Although the orignal Map-reduce system is proprietary, there a few open source implementations such as DisCo [91] and [51], with the latter being used extensively in both industrial and academic settings. In this programming model, users only need to provide two functions, a *mapper* for selecting or filtering data, and a *reducer* for combining or reducing the data. The complexity of dispatching and scheduling jobs is abstracted or hidden from the user by a run-time execution manager.

Beyond Map-Reduce, other abstractions have been introduced. For instance, the Cooperative Computing Lab (CCL) at the University of Notre Dame has introduced All-Pairs and Wavefront [79, 124]. The former is a pattern of computation normally found in biometrics experiments where two datasets must be compared pairwise, while the latter is a distributed dynamic programming pattern often used in economics of genomics. These abstractions not only simplify the construction and execution of certain types of distributed applications, but also tend to be more efficient than naive implementations of the desired workflow.

Overall, all of these distributed computing abstractions have been relatively successful at enabling non-expert users to create certain distributed applications while shielding them from the complexity of distributed systems. Due to their limited and simplified programming models, however, abstractions by themselves are not sufficient for the many types of workflows [60, 122]. For instance, it can be difficult to develop an application that requires multiple instances of an abstraction or even to combine different ones in the same workflow.

A few systems approach the problem of constructing distributed workflow applications by combining both DAG-based systems and computing abstractions. For instance, Oozie [89] is a XML-based workflow system that runs on top of the Hadoop Map-Reduce framework and is used extensively for data processing tasks at web service providers such as Yahoo!. Additionally, Kepler has modules for designating particular nodes or tasks for running on Hadoop [117]. Despite this hybrid approach, these systems still lack expressive programming interfaces and ease-of-use.

2.3 Programming Languages

In response to these deficiencies, there has been a surge in research aimed at developing new programming languages and frameworks to bridge the gap between workflows and abstractions, while still providing the user friendly programming interfaces. Some of these languages build on top of abstractions, while others operate directly on workflow systems. The compiler toolchain proposed in this dissertation is a part of this current research trend and attempts to tackle the challenge of programming distributed workflows.

Pig [88] and Sawzall [93] are two languages that provide a high-level interface to Map-Reduce [35]. The former targets the open source Hadoop platform, while the latter runs on the Google's proprietary system. Both of these languages provide a simplified programming model composed of datasets and functions that is presented as new declarative programming languages with SQL-like syntax [28]. Cascading [25] is a Java library built on top Hadoop that allows users to explicitly construct dataflow graphs in order to program data-parallel pipelines that run on Hadoop's Map-Reduce. FlumeJava [29] is another Java library that runs on top

of Hadoop and also supports constructing data processing pipelines by performing operations on a set of parallel collections provided by the library. Unfortunately, due to the nature of the Hadoop platform, it can be difficult to integrate legacy or external software. Moreover, since these frameworks are tightly tied to the Map-Reduce abstraction, the user is constrained in the types of workflows they can effectively specify.

Dryad [60] is another workflow system where users develop applications through the construction of DAGs. Because the work of building a workflow graph is rather low-level and complex, the authors of Dryad suggest the use of various higher-level tools such as DryadLINQ [59]. This programming construct takes advantage of the LINQ programming idiom in Microsoft's .NET system to allow the specification of Map-Reduce type workflows using a single LINQ [76] expression. Another language built on top of Dryad is SCOPE [26], which is a declarative scripting language where programs are written in a variant of SQL. Like Pig and Sawzall, these Dryad-based languages are tied to their distributed computing platform and thus are limited to the Map-Reduce programming model.

Swift [120, 126] also tackles the problem of specifying diverse scientific workflows, but does so by providing a general purpose programming language complete with a data type system. In Swift, users construct data structures representing their input and output data and specify functions that operate on these structures in a new custom programming language. This specification is then compiled into a set of abstract computation plans which is processed by the CoG Karajan [116] execution engine which works in conjunction with the Swift run-time system and Falkon [97] to execute the plans on loosely-coupled distributed systems such as Condor.

GEL (grid execution language) [71] is another programming language similar to Swift. It requires users to define programs to run and what order to execute the tasks, while handling data transfer and job execution for the users. Unfortunately, it also requires users to explicitly state which jobs are parallel and which ones are not, rather than determining these from data dependencies as in Swift and the proposed compiler toolchain.

GRID superscalar [104] demonstrates the use of an imperative programming language to implicitly construct workflows. In the GRID superscalar programming environment, users utilize either C/C++ or Perl in conjunction with CORBA IDL specifications of the tasks to automatically generating a task data-dependent workflow graph. This workflow generation an execution is accomplished using a run-time library which dispatches tasks in a master-worker paradigm.

Skywriting [82] is an interpreter-based approach to programming cloud applications that utilizes a purely-functional programming language based on Javascript. It relies on futures [5] and lazy evaluation to implicitly extract parallelism from the workflow script. Because its CIEL execution engine [83], it supports dynamic applications that involve iteration and recursion.

2.4 Compiler Toolchain

In addition to adopting ideas from distributed workflows, abstractions, and programming languages, this dissertation incorporates a variety of techniques and methods from research involving compilers and traditional programming languages to produce a compiler toolchain for data intensive scientific workflows. The proposed compiler toolchain in this thesis is meant to be analogous to common development toolchains such as GCC [106] and LLVM [69].

The first set of contributions involves implementing a few compiler optimization techniques in order to increase the performance of the generated distributed workflows. For instance, like a conventional compiler where we need to manage stack allocation [4, 31] and perform name mangling [98, 99], distributed workflows need to manage and store both *a priori* data and intermediate data. Another important optimization method is instruction selection, which is used to translate an intermediate program presentation to a lower-level form closer to the target platform [45]. When performing instruction selection, the goal is to choose the optimal set of instructions that will yield the best performance in the context of the targeted architecture (e.g. SIMD [102] instructions), rather than lowest common denominator code that is portable. A third compiler technique is data partitioning [57], which is used to minimize communication overhead by group instructions (normally in a parallel loop).

In addition to these capabilities, I also investigate and evaluate graph transformation techniques based on conventional compiler optimization methods. One such optimization method is software pipelining [65], where code segments are scheduling in such a way that they can interleave based on data dependencies. Another optimization is unrolling instructions to overcome dispatch overhead [12, 38]. Related to this is function or method inlining, which is also used to mitigate runtime lookup or dispatching [52]. All of these common optimizations are regularly performed on most modern compilers and help increase execution performance in traditional applications.

Distributed computing researchers have only recently begun to utilize similar techniques to optimize the performance of DAG-based workflows. For instance, clustering or transforming a directed graph to reduce dependencies [63] was ex-

plored by the Pegasus project on the Montage and Tomography applications [103]. Some researchers have utilized flowcharts as models to optimize workflows [37], while others have focused on run-time transformations of workflows to improve scheduling and resource allocation [94]. In this dissertation, I concentrate only on compiler optimization techniques that operate on a static DAG representation of a scientific workflow.

In addition to incorporating compiler optimizations, my dissertation also implements a few common toolchain utilities. The first set of tools is a pair of linkers [10] which can be used to package individual applications that are a part of the workflow or to organize the generated workflow DAG as a whole. In other words, these software tools enable static linking [33] of executables and workflows in a manner similar to `crunchgen` [61] and `statifier` [115]. Unlike these tools, however, the linkers presented in this dissertation do not require access to the original source code, and they can package data and environmental settings in addition to executables and libraries into one self-contained application unit. The goal of these linker tools is to encapsulate individual executables (or the entire workflow) with their dependencies (i.e. libraries, configuration files, environment variables) and in order to minimize the complication of distributing dynamically linked [55] applications in a heterogeneous execution environment.

Likewise, I also present a pair of profiling utilities to analyze and monitor the execution of the workflow. These programs are distributed workflow analogs of the traditional `gprof` [50] application found in GCC and the `condor_status -better-analyze` command provided by Condor. Because profiling and monitoring workflows is a complex and ongoing research field [40], I primarily focus on demonstrating the utility of these tools and how they improve the experience of

developing and debugging distributed workflows. The objective here is to provide a set of tools that can parse, display, and export the provenance [8] information generated by the workflow both during and after its execution in an user-friendly manner such that the researchers can monitor and debug their distributed applications.

Finally, this dissertation also modifies and augments the Makeflow workflow manager to support additional run-time capabilities. For instance, garbage collection [15, 121], which typically involves a variety of algorithms for allocating and managing resources such as memory is implemented to tackle the problem of intermediate files exhausting filesystem resources. Additionally, a method of annotation similar to that found in OpenMP [34] and Cilk [13] is used to allow users to specify resource constraints to be propagated. This information is utilized by Makeflow to allow users to take advantage of platform specific options such as Condor ClassAds. Furthermore, Makeflow is modified to make it aware of nested invocations and to manage environmental variables in a manner more consistent with the traditional Make utility [43]. Note, that I avoid addressing any complications with recursive make [77] and only focus on nested configurations in my dissertation.

Overall, the proposed compiler toolchain shares a few of the important features present in these many projects. As noted previously, the compiler toolchain utilizes the DAG-based workflow engine as its run-time system. Because the Weaver compiler does not force users to define workflows in terms of graph nodes and links, it is most similar to DryadLINQ, Swift, and FlumeJava in providing a high-level programming interface. Like FlumeJava, but unlike Swift, Weaver builds on top of an existing programming language, Python, rather than introduce a new

one. This takes advantage of Python's user-friendliness and allows programmers to utilize the plethora of existing Python software. Likewise, Weaver is not restricted to a single programming construct as in DryadLINQ, Pig, and Sawzall, but encompasses a whole library of components that form a domain-specific language for distributed computing. Furthermore, the compiler toolchain also applies techniques found in traditional compilers and programming languages such as optimizations and garbage collection. All of these features combine into a compiler toolchain that enables users to rapidly and effectively construct data intensive scientific workflows.

CHAPTER 3

COMPILING WORKFLOWS

The central component of a programming toolchain is the compiler. In order to evaluate the application of compiler techniques to distributed workflows, I implemented the Weaver [23] workflow compiler that enables computational researchers to construct distributed data intensive scientific workflows in the Python programming language.

To construct a distributed workflow using Weaver, the user programs a specification in Python that utilizes various `Dataset`, `Function`, `Abstraction`, and `Nest` components provided by the Weaver programming interface. Once the specification is complete, the user processes the script using the Weaver compiler which generates a workspace (sandbox directory) that contains a directed acyclic graph (DAG) enumerating each task in the workflow and their relationship with each other and any other files materialized by the compiler during compilation. The generated DAG is passed to a workflow manager whose job is to schedule the tasks specified in the generated DAG by dispatching the jobs to a distributed execution engine. This chapter briefly summarizes the programming interface provided by the compiler, examines the execution model of the compiler, and discusses the components of the entire software stack.

3.1 Programming Interface

Weaver provides a simple, though restricted, programming model that consists of `Datasets`, `Functions`, `Abstractions`, and `Nests` as shown in Table 3.1. These concepts are the fundamental building blocks of the Weaver application programming interface (API) and are implemented as a custom Python package consisting of modules, classes, and functions that end users combine and extend to define their distributed scientific workflows.

TABLE 3.1

WEAVER PROGRAMMING INTERFACE COMPONENTS

Component	Summary
Datasets	Collections of data objects that represent physical files.
Functions	Specifications of executables used to process data.
Abstractions	Patterns of execution that define how functions are applied to datasets.
Nests	Execution context consisting of a DAG and namespace.

The first component of the Weaver programming interface is the idea of a `Dataset`. Data intensive scientific workflows typically involve processing and analyzing a repository of experimental data that is stored as files on the filesystem. In the Weaver, collections of such data are represented by `Dataset` objects where

each element's string (i.e. `__str__`) method returns the filesystem location of a particular item's data. This convention means that a `Dataset` in Weaver can be a Python list, set, generator, and any other object that implements Python's iteration protocol.

TABLE 3.2

WEAVER DATASETS

Dataset	Description
`Glob`	Collection of files based on path expression.
`FileList`	Collection of files stored in a text file.
`SQLDataset`	Collection of data from a SQL database.
`Query`	ORM selection and filtering function for Weaver `Datasets`.

Table 3.2 provides a list of the `Dataset` objects provided by Weaver. The first `Dataset` constructor provided by Weaver, `Glob`, allows users to select a collection of files based on a path expression such as `*.dat`, while the second `Dataset`, `FileList`, specifies that the data files are enumerated in a text file. The third `Dataset`, `SQLDataset`, provides users a straightforward mechanism for extracting data from a SQL [28] database such as MySQL [84] and transparently materializing the data as physical files. All of these `Datasets` can be filtered into smaller subsets using Weaver's `Query` function, which allows users to perform selection

and filtering operations on `Datasets` using an ORM [62] expression language similar to SQLAlchemy [105].

The second component of Weaver's programming interface is the `Function`. In most scientific workflows, an *ensemble* of executables are used to process and analyze a set of data. Weaver accounts for these applications by providing the notion of a `Function` specification object that defines the location of the executable and its command line interface. This means that each `Function` specifies information such as the path to the executable and how arguments to the `Function` are to be formatted to generated a *shell command* that can be executed to perform the desired operation. Like objects in a `Dataset`, each `Function` also corresponds to an object on the filesystem.

TABLE 3.3

WEAVER FUNCTIONS

Function	Description
`Function`	Base `Function` object constructor.
`ParseFunction`	Convenience wrapper that constructs `Function` and sets command format.
`ShellFunction`	Constructs `Function` out of shell script specified as string.
`PythonFunction`	Constructs `Function` from inline Python code.
`Pipeline`	Combines multiple `Functions` into a single meta-`Function`.

As shown in Table 3.3, Weaver provides a set of custom Python components designed to expedite and simplify the specification of workflow `Functions`. The first item in the table is the base object constructor from which all other constructors are derived and requires the user to specify the location of the executable and the *command format* specifying how the *shell command* should be arranged. `ParseFunction` is a utility wrapper that will construct a `Function` based on a string template and will automatically set the location and *command format* for the user. It is used internally by `Abstractions` when parsing the `Function` argument and is provided to the user for convenience. The next two `Functions`, `ShellFunction` and `PythonFunction` allow the user to embed or inline shell and Python code respectively. This means that rather than having to create external scripts, the user can simply place the scripts inside the main workflow specification and the compiler will automatically materialize a script file for the user. The final `Function` is `Pipeline`, which enables users to combine multiple `Functions` into a single meta-`Function` that behaves as a normal `Function`.

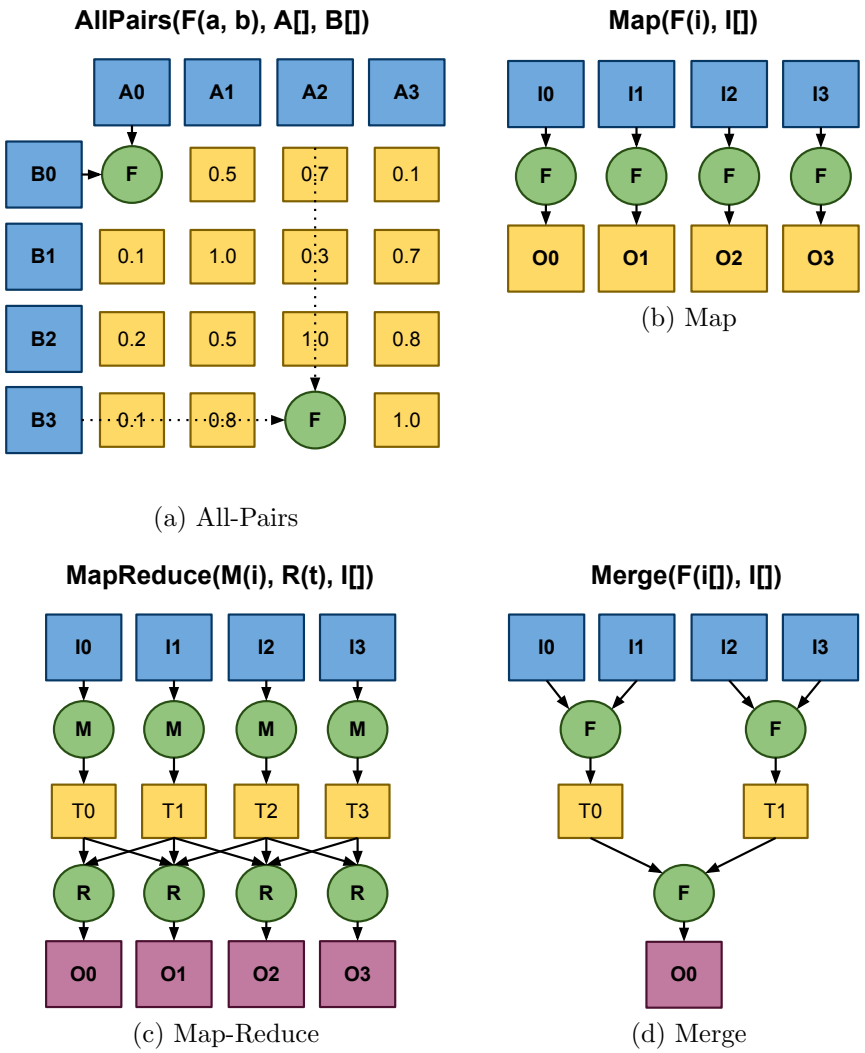
The third component in the programming interface is `Abstractions`. These are high-order functions that specify a specific pattern of computation with a precise set of semantics. In Weaver, `Abstractions` are basically lazily evaluated [5] `Datasets` that take `Function` and `Dataset` arguments and specify how the `Functions` are to be applied to the input `Datasets` to produce the output `Dataset`. As such, the result of one `Abstraction` can be passed as an argument to another `Abstraction` or to a `Function` to coordinate a sequence of operations. This means that rather than explicitly forming a graph of tasks, users implicitly construct their workflow DAGs by passing `Datasets` as inputs to `Abstractions` and `Functions`.

TABLE 3.4

WEAVER ABSTRACTIONS

Abstraction	**Description**
AllPairs(`function, inputs_a, inputs_b`)	Apply `function` to all pair-wise combinations of `inputs_a` and `inputs_b`.
Map(`function, inputs`)	For each `input` in `inputs`, apply `function`.
MapReduce(`mapper, reducer, inputs`)	For each `input` in `inputs` apply `mapper`, sort intermediate outputs, and then apply `reducer`.
Merge(`function, inputs`)	Use `function` combine `inputs` by using a parallel reduction.

Table 3.4 summarizes the four `Abstractions` provided by the Weaver compiler. The first `Abstraction` is `AllPairs` [78], which is a pattern of computation where each member of one dataset is compared to each member of a second dataset to produce a matrix that contains the resulting scores for each comparison. The second `Abstraction` is `Map`, which involves applying a `Function` to each item in a `Dataset` to produce a new output `Dataset`. `MapReduce` [35] is a computational pattern that first applies a *mapper* `Function` to the input `Dataset` to generate an intermediate collection of key-value pairs that are shuffled, sorted, and then processed by the `reducer Function`. The final `Abstraction` is `Merge` which performs a parallel reduction [64] on the input `Dataset` using a specified merge `Function`. As illustrated in Figure 3.1, all of the tasks generated by these `Abstractions` exhibit data independence and thus can be scheduled concurrently and executed in parallel.



(a) All-Pairs

(b) Map

(c) Map-Reduce

(d) Merge

Figure 3.1. Weaver Abstractions

The structure of the four `Abstractions` *provided by Weaver are illustrated in this Figure. In each pattern of computation, the processing of the data can be performed independently and thus can be scheduled to execute in parallel or concurrently.*

The final component in the Weaver programming interface is the notion of a `Nest`. In Weaver, all workflows consist of a workspace and a directed acyclic graph. The workspace serves as a storage area for any intermediate and output workflow artifacts, while the DAG encodes the relationships between tasks in the workflow. `Nests` are Weaver objects that represent both a workspace and DAG. Whenever an `Abstraction` is processed, it is done so in the context of a particular `Nest`, which captures any tasks produced by the `Abstraction` being executed. Figure 3.2 illustrates the structure of Weaver `Nest` object. As can be seen, a `Nest` consists of both a workspace and a DAG.

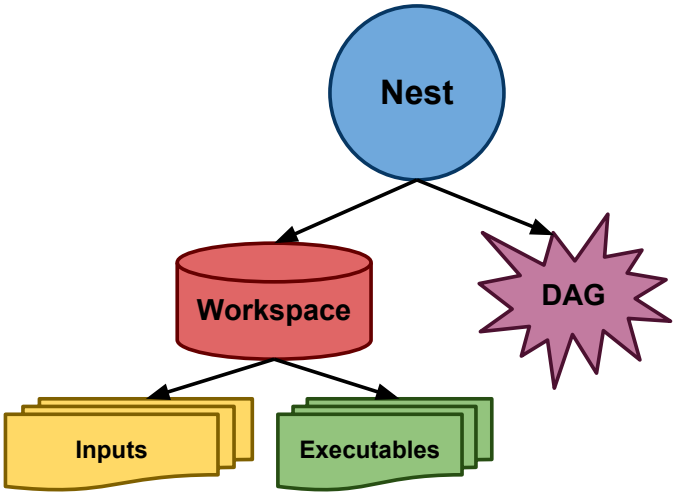


Figure 3.2. Weaver Nests

In Weaver, a `Nest` *is a conceptual object that consists of a namespace (e.g. workspace on the filesystem) and a directed acyclic graph that contains the tasks in the workflow.*

One way to understand the `Nest` concept is to consider that a workflow typically has two key features that distinguish it from other workflows: **(1)** a namespace and **(2)** a DAG. The namespace of the workflow determines the environment in which the workflow is to be executed. Typically this namespace corresponds to workspace on the filesystem and thus can be mapped to a specific sandbox. In addition to this namespace, a workflow also consists of a directed acyclic graph that specifies the tasks to be performed and how they are related to one another.

Taken together, both the namespace and the graph combine to uniquely identify a single workflow. It is possible, for instance, that a single namespace would contain multiple DAGs or for a single DAG to exist in multiple namespaces. To properly identify a specific workflow, then, we must use the combination of both the namespace and the DAG. In Weaver, each `Nest` object is associated with a particular namespace and a single DAG. Therefore, each `Nest` maps to a specific workflow. That is, a workflow in Weaver is a represented by a `Nest`. Whenever we compile a workflow using Weaver, we are constructing a `Nest`.

To utilize the Weaver compiler, users employ the `Dataset`, `Function`, `Abstraction` and `Nest` components described here to specify their distributed workflow. The programming interface is restrictive in that it requires that all data and functions to be represented on the filesystem, but is flexible enough to support a variety of computational patterns. The compiler comes with a collection of components for the user to utilize, but also allows developers to extend these modules and even add their own. A more in-depth discussion of the Weaver API, along with example source code, is provided in Appendix A.

3.2 Workflow Compiler

As mentioned earlier, the workflow language is implemented as a domain-specific language on top of Python. This means that rather than having its own lexer and parser, Weaver depends on the language facilities of the Python interpreter. When writing Weaver scripts, the user is basically programming in a restricted subset of Python that primarily consists of invoking and utilizing the various `Datasets`, `Functions`, and `Abstractions` components discussed previously. The advantage of this is that users get to leverage their familiarity with Python instead of having to learn a new programming language.

To construct a workflow, users simply utilize the Weaver components in a Python script to specify their distributed workflow. In addition to the Weaver library and the standard Python library, users may employ any available third party Python module such as NumPy or SciPy [87] to construct their workflow. Once the workflow application is completely specified, the script is passed to the Weaver compiler for processing.

As shown in Figure 3.3, the output of the Weaver compiler is a sandbox directory that contains the DAG file detailing the tasks scheduled by the compiler. If necessary, any scripts or files materialized by the compiler (e.g. `ScriptFunctions` or `PythonFunctions`) are placed in this sandbox. As such, this sandbox directory represents the `Nest`'s workspace component, while the DAG file generated by the compiler corresponds to the `Nest`'s workflow graph component. Together, the sandbox and DAG file represent a single unique workflow.

When a script is passed to Weaver, the compiler begins processing the specification by initializing the output sandbox which serves as the default location for the compiler's output. This includes creating the directory if it does not exist

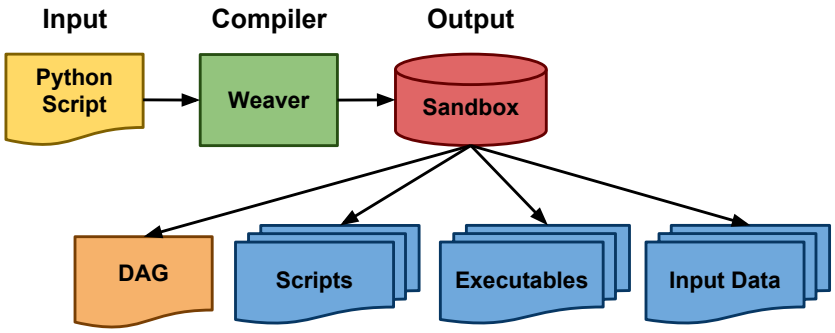


Figure 3.3. Weaver Compiler

In Weaver, a workflow is specified using the Python library components provided by the compiler. This specification is passed to the compiler that evaluates the Python script and generates a sandbox directory that contains a DAG file enumerating the tasks necessary to execute the desired workflow and any files materialized during compilation.

and setting up the `Stash` structure which is described in Section 4.1. After this, the compiler configures an initial Python environment by importing the various components of the framework into the global interpreter namespace and setting required paths. After this the compiler evaluates the specified Python script using Python's `execfile` function, which reads in the Python script and evaluates it using the environment setup by the compiler.

The side-effect of this evaluation is the generation of the workflow DAG and any necessary materialized files. That is, in order to generate a Weaver workflow, it is necessary to evaluate the Weaver script in a specially constructed Python environment. The toolchain includes a Python script, `weaver`, that sets up the sandbox, configures the environment, and evaluates the Weaver specification as described in the previous paragraph and is considered the compiler.

It should be noted, however, that it is possible to utilize the Weaver library components without this compiler (i.e. execute the script using Python directly

rather than with `weaver`) and still have it generate proper workflows. In fact, evaluating or executing the `Dataset`, `Function`, and `Abstraction` components directly in Python will work as long as an initial `Nest` object is setup as the current context manager. For the most part, however, it is recommended that users generate workflows with the `weaver` compiler script which ensures the environment is properly configured.

During the evaluation of the workflow script, the compiler tracks the tasks generated by the `Abstractions` and `Functions`. Whenever users invoke one of the Weaver `Abstractions` in the Python script, `Functions` are applied to `Datasets` in the pattern proscribed by the `Abstraction` and a sequence of tasks in the form of `(abstraction, function, command, inputs, outputs, options)` tuples is scheduled with the current `Nest`.

As noted previously, in Weaver, `Abstractions` *are* `Datasets`. In fact, the current implementation has `Abstractions` as a child class of the `Dataset` class. This is because in Weaver, `Abstractions` generally behave as `Datasets` (i.e. users pass them as inputs to `Functions` or other `Abstractions`), except that they also schedule some computation (`Datasets`, on the other hand, do not schedule any computation). Because of this, `Abstractions` are *memoized futures* [5] that **(1)** are lazily evaluated and **(2)** cache their results. This is important to consider because if an `Abstraction` is not iterated over (i.e. used as an argument to another `Abstraction` or `Function`), then it will never be computed and thus never scheduled in the workflow.

To prevent the situation of using an `Abstraction`, but not generating any tasks during compilation, Weaver registers each `Abstraction` with the `CurrentNest` during initialization of the `Abstraction` instance. When the `Nest` goes to compile

the workflow, it will iterate over every `Abstraction` associated with it to ensure that every `Abstraction` is computed and scheduled. Since the results of iterating over a `Dataset` are cached, we will never schedule tasks more than once because the `Dataset` will only be generated once.

If the user utilizes the `Nest` constructor in the workflow specification, then a new sub-workflow is initialized and subsequent tasks are associated with this new `Nest`. This continues until the `Nest` is out of scope, and then the previous `Nest` is restored. Just as `Abstractions` are registered with the `CurrentNest`, child `Nests` are registered with their parents. This is done to ensure that all of the `Abstractions` defined in the child `Nests` are appropriately compiled.

When the `weaver` compiler is finished evaluating the Weaver script, it will then call the `compile` method of the `CurrentNest`. This will in turn perform the cascading series of compilations described above. Besides iterating over `Abstractions` and compiling sub-`Nests`, the compiler will also perform some optimizations. For instance, rather than having a single DAG node responsible for a single task, the user may wish to aggregate a group of tasks into one single super DAG node. This is an example of clustering which was shown by the Pegasus project to have significant performance benefits on the Montage and Tomography applications [103]. Another possible optimization technique is the use of instruction selection to take advantage of native optimization tools. These optimizations and others implemented by the compiler are discussed in further detail in Chapter 4. Once everything is compiled, the compiler will then emit the computed DAG in a format suitable for the target workflow manager.

In summary, a directed acyclic graph of tasks is generated as a side-effect of the evaluation of the Weaver script that utilizes the `Dataset`, `Function`, and

`Abstraction` components of the Weaver programming interface. All of these tasks are tracked by a `Nest` object, which will perform various optimizations on the graph and finally emit a DAG representing the specified workflow. A more detailed examination of the Weaver's implementation is provided in Appendix B.

3.3 Workflow Manager

As mentioned previously, the Weaver compiler generates a sandbox directory containing a DAG and various files required for proper execution of the workflow. To actually execute the workflow, the user must pass this workspace and DAG to a workflow manager which will use the sandbox as the storage area for the outputs of the workflow and any intermediate workflow artifacts generated during execution and will parse the DAG to generate tasks to execute on various distributed execution platforms.

Currently, the compiler only supports the Makeflow [3] workflow manager although the initial version [21] did support Condor's DAGMan system [1]. The decision to only support Makeflow was done because some critical aspects of the dissertation required modifying the run-time system to support some of the desired features of the toolchain. Because of my familiarity with Makeflow, I decided to only target it and add the features the toolchain required to it. These modifications are discussed in detail in Chapter 7.

When a workflow script is compiled with Weaver, a Makeflow DAG is generated in the sandbox directory. This DAG contains rules similar to those found in a regular UNIX Makefile [43] that describe tasks in terms of the inputs and output dependencies and the command used to accomplish the desired task. These rules are used by Makeflow to form an internal directed acyclic graph of the entire

workflow. In this graph, the nodes are the data to be processed and the tasks to be executed with this data, and the links are the relationships between the tasks and the necessary input and output files. By forming this directed graph, Makeflow can accurately determine which tasks depend on others and schedule the work appropriately to optimize concurrency and parallelism.

TABLE 3.5

WEAVER AND MAKEFLOW DAG EXAMPLE

Weaver Source	**Makeflow DAG**

`jpgs = [str(i) + '.jpg' for i in range(1000)]`
`conv = ParseFunction('convert {IN} > {OUT}')`
`pngs = Map(conv, jpgs, '{BASE_WOEXT}.png')`
	`0.png: 0.jpg /usr/bin/convert`
	`/usr/bin/convert 0.jpg 0.png`
	`1.png: 1.jpg /usr/bin/convert`
	`/usr/bin/convert 1.jpg 1.png`
	`2.png: 2.jpg /usr/bin/convert`
	`/usr/bin/convert 2.jpg 2.png`
	...
	`999.png: 999.jpg /usr/bin/convert`
	`/usr/bin/convert 999.jpg 999.png`

An example of the relationship between the user-defined Weaver workflow script and the generated Makeflow DAG is shown in Table 3.5. In this simple

example, we wish to convert a thousand JPG images to PNG format. We can specify this workflow in three lines of Weaver as shown on the left side of the table. If we process this workflow specification with the Weaver compiler, we get the Makeflow DAG displayed on the right side of the table below. For each JPG file, we created a rule consisting of the output PNG file, the input JPG file, the `convert` executable, and the *shell command* required to perform the task. Note the listing on the right is only an abbreviation of the Makeflow DAG, as the whole DAG would be two thousand lines, which is much longer than the three line Weaver specification. As can be seen, Weaver enables concise specifications of large data intensive workflows and thus improves the expression of such workflow applications.

This is the one of the key reasons for compiling workflows rather than constructing them directly. With a high-level language such as the Weaver DSL, it is possible succinctly specify a complex workflow consisting of thousands to millions of tasks in a very few lines of code. Constructing the DAG manually would be tedious and error-prone, while using ad-hoc scripts would become difficult to manage in the long-term. Having a formalized method of generating workflows, as provided by the Weaver compiler, enables rapid and reliable development of these data intensive applications. Additionally, as we will see in Chapter 4, compiling workflows also provides an opportunity to perform optimizations that can boost the performance of the workflows.

The complete Weaver software stack is composed of three layers as shown in Figure 3.4. The first layer is the Weaver compiler framework which is used to translate a workflow specification in Python into a Makeflow DAG. The second layer is the Makeflow workflow manager which processes the DAG and dispatches

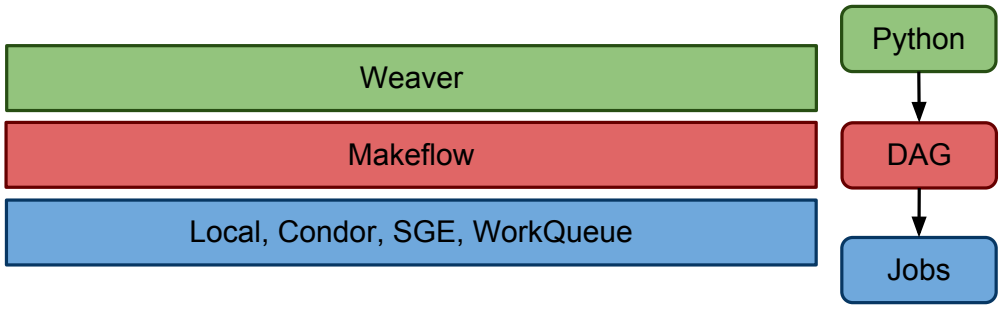


Figure 3.4. Weaver Software Stack

To create and execute workflows using this toolchain, users write workflow specifications in Python and then compile them using Weaver. This will generate a workspace and a DAG containing the tasks necessary for accomplishing the workflow. In order to execute the workflow, the workspace and DAG is passed to the Makeflow workflow manager which will create batch jobs on a variety of execution platforms.

jobs to the third layer, the various distributed execution engines. As a portable DAG-based workflow manager, Makeflow provides the ability to utilize different execution engine such as Condor [113], Sun Grid Engine (SGE), WorkQueue [22], and local Unix processes. To perform workflow execution, Makeflow internally employs the master-worker paradigm [72] to perform task scheduling and resource allocation.

Because Weaver generates Makeflow DAGs rather than directly scheduling for a specific execution engine, users of the framework can easily take advantage of multiple execution environments by simply selecting the appropriate platform at run-time. This flexibility allows users to adapt to the resources available to them without having to modify their workflow specification. Additionally, Makeflow utilizes a transaction journal to record the progress of a workflow. This journal is normally stored as a plain text file in the workflow's workspace and can be used

to collect provenance information such as the number of tasks failures, attempts, execution times, and more. The log also enables Makeflow to resume or restart a failed workflow without rescheduling already completed tasks. Batch system specific logs such as a Condor log file are also stored in the workspace and can also be used to collect provenance information.

Overall, this system architecture is similar to other workflow systems such as Swift and Pegasus. For instance, Swift relies on CoG Karajan execution engine [116, 126] to dispatch tasks and perform resource allocation, while Pegasus relies on Condor DAGMan [1, 36] for distributed execution. In this case, the toolchain presented in this dissertation depends on the Makeflow workflow manager to perform the actual execution of the workflow. Because Makeflow supports a variety of distributed execution platforms, this means that Weaver workflows can be easily ported to and executed on different distributed systems.

Finally, the power and utility of the Weaver's execution model is that the compiler allows for users to rapidly construct and configure workflows in a high-level language (Python). This is important because it is not always clear what the best decomposition of a workflow should be or what elements are required. Using the Weaver compiler, users can rapidly prototype and generate their distributed workflows in a concise and reliable manner. With Makeflow's support for local and distributed systems, these workflows can be tested locally on a multi-core system and then later executed on different distributed platforms when the workflows have been debugged.

CHAPTER 4

OPTIMIZING WORKFLOWS

As with traditional compilers, Weaver supports a few optimization techniques that can be used to increase the performance or run-time characteristics of the generated workflows. This chapter discusses four optimization methods that were implemented by the Weaver workflow compiler and examines their effect on the performance of a few sample benchmarks. These methods were inspired by techniques from traditional programming language compilers and are summarized in Table 4.1. As noted in the table, each optimization corresponds to a traditional compiler technique and has its advantages and disadvantages. In addition to discussion and testing these optimizations, this chapter also includes analysis for when these methods are appropriate for achieving improved performance. Overall, these optimization techniques can yield significant performance increases under the right circumstances and provide evidence of the effectiveness of applying certain traditional compiler methods to DAG-based workflows.

TABLE 4.1

COMPILER OPTIMIZATIONS OVERVIEW

Optimization	**Analogy**	**Advantages**	**Disadvantages**
Structured Allocation	Name mangling	Prevents inode exhaustion, keeps sandbox clean	Slower compilation, negligible performance increase
Instruction Selection	Instruction selection	Take advantage of optmized implementation	Lack of availability, limited interface
Hierarchical Workflows	Data partitioning	Increase concurrency, interleave execution	Explicit management, barriers
Inline Tasks	Loop unrolling	Minimize dispatch time	Namespace complications, break constraints

4.1 Structured Allocation

The first optimization method attempts to address the problem of intermediate files. During the execution of a large workflow, it may be necessary to generate many intermediate output files. A problem occurs when these intermediate files are naively placed in a single directory: many filesystems either have limits to how many files can be placed in a single folder or greatly degrade performance with after a certain threshold. To work around these filesystem limits and to avoid performance degradation, the workflow compiler must have an intelligent way of managing intermediate output files.

In a sense, we need a form of *name mangling* [98, 99] combined with *stack allocation* [4, 31] for our workflows. In a conventional compiler, name mangling is used to uniquely encode programming entities with the same identifier but different namespaces, while stack allocation involves automatically reserving a fixed amount of storage in a function's reserved region of memory. In distributed workflows, the locations of the intermediate files need to be modified such that filesystem limits are not exhausted. In other words, the compiler must utilize intelligent namespace organization in order to efficiently allocate a storage resources.

In Weaver, the solution to this problem is to allow users to utilize a structure called the `Stash`, which is an object that returns a unique filename for each invocation. This means that whenever a user needs to generate a name for an output file, he may simply call `next(CurrentNest().stash)`, which will return the next unique path from the current `Nest's Stash` (every `Nest` has its own `Stash` structure). The `Stash` enables users to avoid exhausting filesystem limits by spreading the files across a hierarchy of directories, rather than a single folder as shown in Figure 4.1. This is a technique utilized by ROARS [18] to overcome

similar filesystem limits in storing a massive scientific data archive and Parrot [112] for caching remote files.

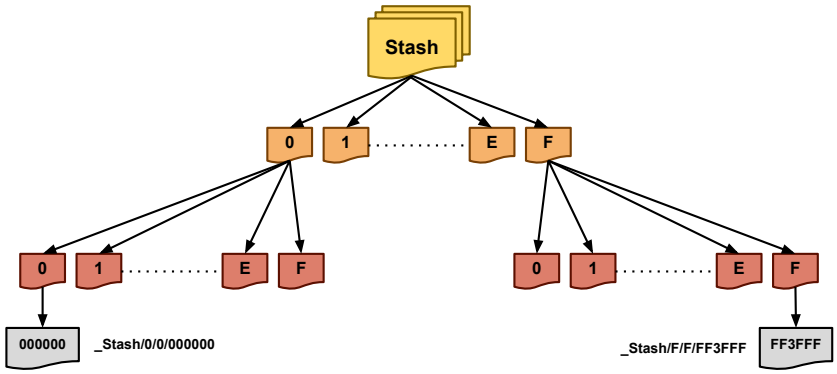


Figure 4.1. Stash Structure

Rather than storing all the files in a single directory, the `Stash` *spreads the files out across a hierarchy of folders. This diagram shows a three level* `Stash`*. In the actual Weaver implementation four levels are used.*

In the default configuration, each `Stash` object contains four levels of directories with the actual files placed in the lowest level. The first level is the root directory and is usually named `_Stash` and placed in the `CurrentNest's` workspace. The next three levels consist of 16 directories on each level, with the folders corresponding to the hexadecimal numbers `0 - F`. As mentioned previously, files are placed in the bottom directory, with the following format: `<L2><L3><L4>XXXX` where `<L2>`, `<L3>`, `<L4>` correspond to the hexadecimal numbers belonging to those levels and `XXXX` is a 4-digit hexadecimal number. The full path of the file then is `_Stash/<L2>/<L3>/<L4>/<L2><L3><L4>XXXX`.

For example, in a four level configuration the first unique `Stash` file path is `_Stash/0/0/0/0000000`. The last file in this setup is `_Stash/F/F/F/FFF3FFF`. This is because the `Stash` limits the number of files in a single directory to 2^{14} (16384 in decimal or 3FFF in hexadecimal) in order to avoid surpassing filesystem limits. In total, a four level `Stash` can support $16^3 \times 2^{14} = 67,108,864$ unique intermediate output files, which, from our experience, is sufficient for most data intensive scientific workflows. If this default configuration is insufficient, the depth of the `Stash` can be modified by simply adjusting the `depth` attribute of the appropriate `Stash object`.

In order to test the effectiveness of the `Stash` structure, I devised a simple workflow as shown in Figure 4.2. The test workflow involves creating a file and listing the contents of the directory into that file for $100,000$ iterations. In the first version of the workflow, we store the all the files in the same directory, while in the second version we explicitly utilize the `Stash` to manage the location of the files for us. Normally, if we do not explicitly set an output file template to an `Abstraction`, then the `Stash` will be used to generate output paths.

```
# 1. Without Stash
Iterate('touch {OUT} && ls > {OUT}', 100000, '{i}.out')

# 2. With Stash
Iterate('touch {OUT} && ls > {OUT}', 100000, '{stash}')
```

Figure 4.2. Stash Benchmark Workflows

This is a simple workflow that creates 100,000 files. Without using the `Stash` *these files will be created in the workflow directory. With the* `Stash`*, these files will be spread across the internal* `Stash` *hierarchy.*

To measure the effectiveness of the `Stash`, I executed the described workflows using Makeflow's local batch system on a 16-core machine and employing a sandbox located on a local filesystem, AFS [56], and NFS [100]. The results of these `Stash` benchmarks are shown in Figure 4.3.

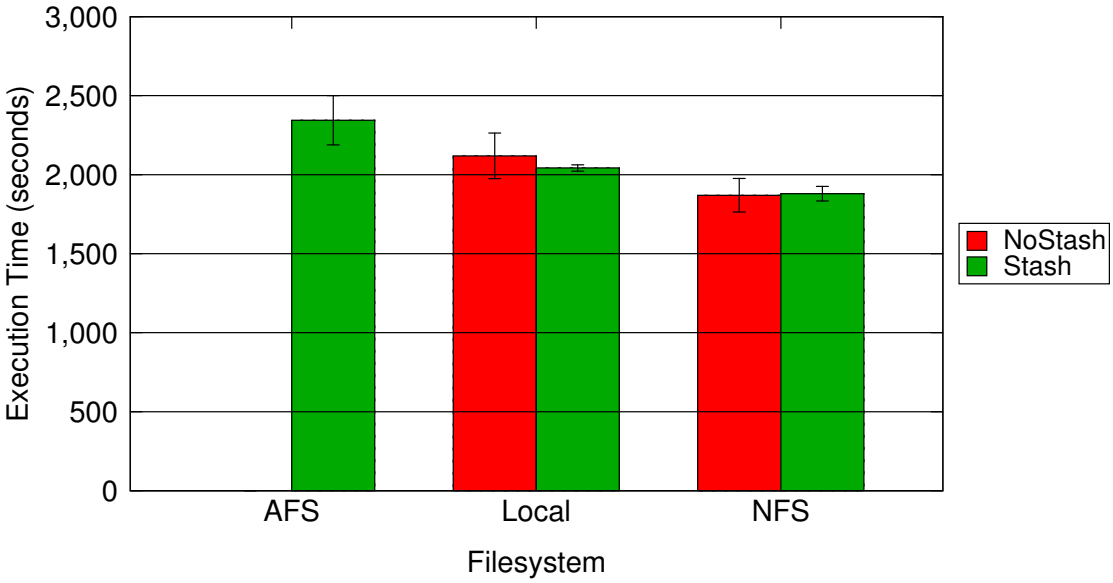


Figure 4.3. Stash Benchmark Execution Times

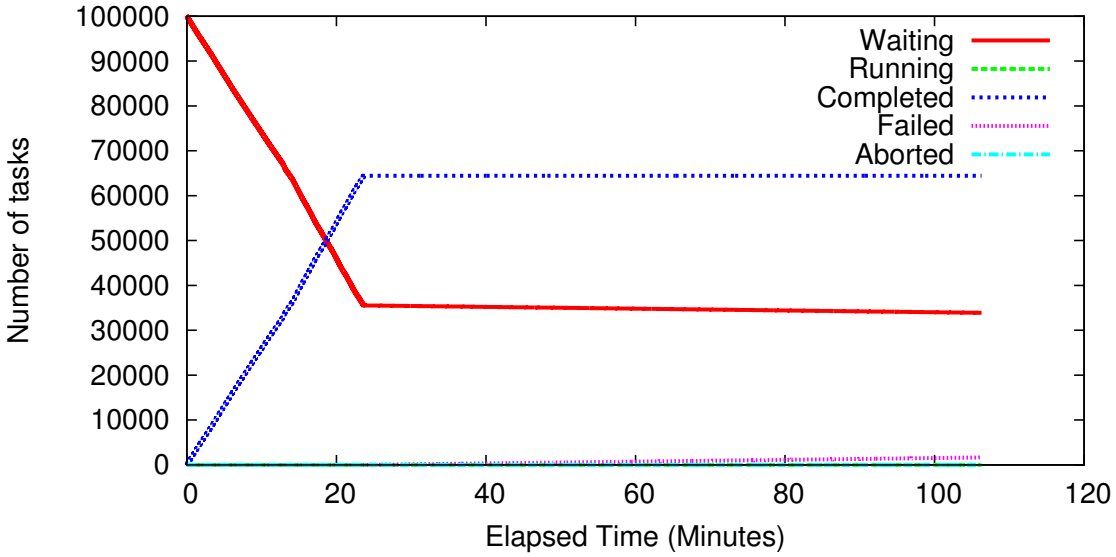
This graph shows the execution times of the iteration workflow with and without the `Stash` *on three types of filesystems: AFS, local, and NFS. For the local and NFS filesystems, the* `Stash` *made little to no difference in performance. However, for AFS, the* `Stash` *enabled the workflow to complete; without it, the iteration workflow would fail.*

In the case of the local filesystem, the `Stash` provided a modest performance increase in terms of execution time. On NFS, the `Stash` yielded a slight perfor-

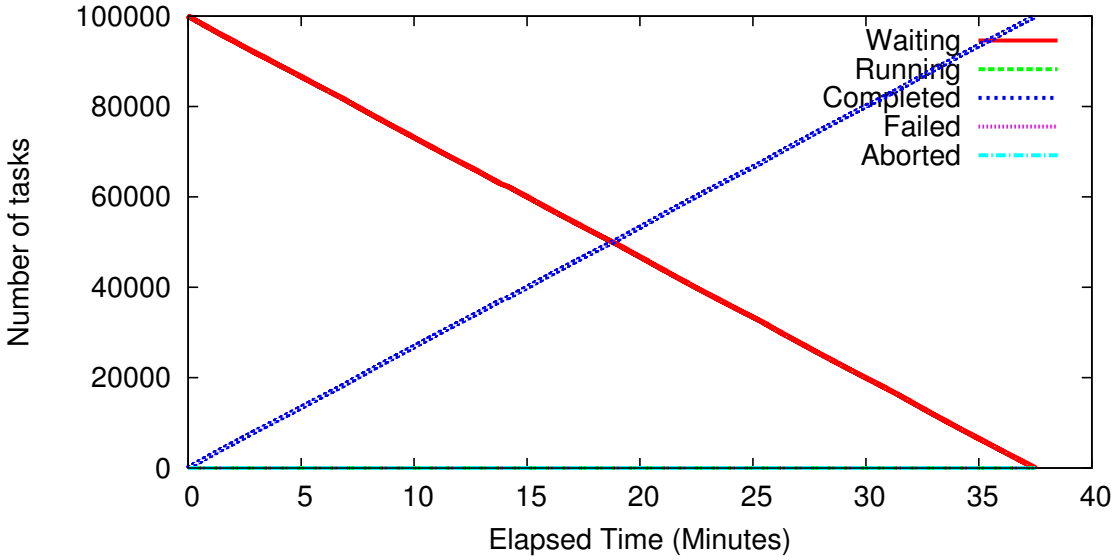
mance decrease. In both cases, however, the execution times for the workflows with and without the `Stash` were within the standard deviation of each other (as indicated by the error bars), meaning the performance differences were insignificant. From the evidence provided by the benchmarks the use of the `Stash` structure did not greatly impact the performance of the workflow on the local filesystem or NFS.

The benchmarks for executing on AFS, though, did produce interesting results. As can be seen in Figure 4.3, there is no bar for the workflow without a `Stash` while using AFS. This is because the workflow could not complete without utilizing the `Stash` when executing on AFS. Figure 4.4 provides sample execution timelines of the workflow with and without the `Stash` on AFS. Without this structure, the workflow stalls around 25 minutes when the workflow has generated around $64,000$ files as shown in Figure 4.4a. This is because AFS has a hard limit of 2^{16} or $65,536$ inodes per directory. Without the `Stash`, each subsequent file creation will fail, and thus block progress of the workflow. Because the `Stash` limits the number of files in one directory to 2^{14} and spreads the files across multiple folders, it allows the workflow to complete as shown in Figure 4.4b.

From these results it is clear that while the `Stash` does not significantly affect the performance of the workflow, it can be critical to enabling workflows on filesystems such as AFS with limited inodes per directory. Because it does not harm performance and makes it possible for workflows with many intermediate files on certain filesystems, Weaver utilizes the `Stash` by default for anonymous or implicit output files. To forgo the `Stash`, users may simply explicitly name all of their output files. As such, the `Stash` is there as a convenience to help users deal with managing many intermediate files, but does not force them to utilize it.



(a) Without Stash



(b) With Stash

Figure 4.4. Stash Benchmark on AFS Sample Timelines

Without the `Stash`*, the iteration workflow stalls after about 64,000 tasks, which is near the limit for the number of files in an AFS directory, and never completes (it was manually aborted after 100 minutes). Using the* `Stash`*, the workflow was able to complete in a little under than 40 minutes while executing from an AFS directory.*

As the issue of managing intermediate files is a persistent problem in large data intensive scientific workflows, Chapter 7 investigates garbage collection as an alternative method of addressing this problem. Fortunately, the compiler is implemented in such a manner that the user may utilize both the `Stash` and garbage collection, just one method, or none at all. The guiding design principle for the Weaver compiler is to provide facilities for improving the performance and execution of the workflow, but to ensure the end user has control of what is utilized. This principle is reflected and reinforced in the optimizations discussed in the remainder of this Chapter.

4.2 Instruction Selection

As discussed earlier, Weaver implements a variety of distributed computing abstractions such as *All-Pairs*. Normally, Weaver `Abstractions` generate a sequence of task tuples in order to implement the appropriate execution pattern while relying on the workflow manager to assemble a DAG and determine proper execution order. This allows for the specification of sophisticated workflows consisting of high-level abstractions that are completely agnostic of the underlying execution engine. One can view the abstraction primitives provided by Weaver, then, as generic operations that would work on any distributed execution platform.

Unfortunately, these generic operations are not necessarily the most optimal or efficient implementations of the particular pattern. One reason is that the Weaver programming model requires input and output data to be manifested as physical files. In workflows that involve many short running applications, this model will yield an inefficient implementation since each data record will need to be instantiated on the filesystem and each application will appear as a separate

task in the DAG. Depending on the execution engine, the latency for each job start up can greatly diminish the performance of such a workflow.

Another reason why the generic implementations may be non-optimal is that some abstractions require intimate knowledge of the underlying distributed system to be effective. For instance, the *Map-Reduce* implementation presented by Google is successful not only because of the data parallel task scheduling but also because of its ability to take advantage of data locality [35].

Because there already exists optimized tools that implement some of the abstractions provided by Weaver and because the generic implementations may be non-optimal, Weaver allows users to perform *instruction selection* [45]. That is, they can easily specify which version of the abstraction to utilize during compilation. To use a optimized native version of an abstraction, the user sets the `native` keyword argument in the abstraction's constructor to `True`. If the native version is available for that abstraction, then Weaver will use that optimized tool instead of a generic version. That is, instead of generated a complete DAG to implement the desired pattern, Weaver will schedule a single task that calls the native implementation to perform the abstraction.

Figure 4.5 demonstrates this transformation for the *All-Pairs* abstraction. Instead of generating all of the tasks that would implement this pattern, the compiler simply generates a single task that utilizes the optimized tool to accomplish the desired computation. Beyond utilizing the optimized native tool, this transformation also greatly reduces the size of the DAG, which can yield small performance increases due to reduce scheduling overhead on the part of Makeflow.

This ability to choose between a generic and specialized operation is similar to the use of SIMD [102] instructions with computer processors. In a conventional

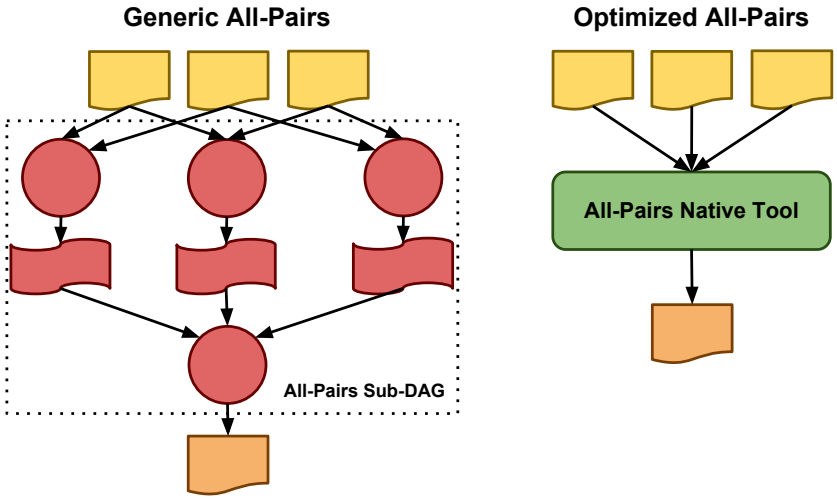


Figure 4.5. Instruction Selection

Normally, abstractions are implemented by generating a set of tasks that when executed will perform the desired computation. When using an optimized native tool, the compiler can avoid generating all of these tasks and instead schedules a single task that utilizes the optimized tool to perform the abstraction.

compiler, operations are compiled using the lowest common denominator instruction set for a particular processor architecture. However, if the user requests an architecture specific optimization, such as SIMD instructions, the compiler will output optimized program code that takes advantage of the hardware [45]. Weaver behaves in a similar manner. By default, it generates DAGs with generic implementations of any requested abstraction. If the user specifies the use of a native tool, then Weaver will forgo the generic version and instead use the optimized tool. As shown in Figure 4.5, a native tool can greatly reduce the size of the DAG since it no longer needs to implement the whole abstraction as a set of task rules and instead uses the specialized software as a single optimized task.

To demonstrate the performance gains possible in using a native implementation over a generic DAG, I wrote two workflows that performed an *All-Pairs* on a

set of `BIT` iris templates from the BXGrid [17] biometrics repository as shown in Figure 4.6. Using Makeflow's WorkQueue batch system and a pool of 100 workers, I varied the size of the input dataset from 10 to 1,000 iris templates and executed both the generic and optimized workflows multiple times.

```
1  # 0. Common code
2  all_bits = Glob('{0}/*.bit'.format(CurrentScript().arguments[0]))
3  bits_set = Query(all_bits, limit=int(CurrentScript().arguments[1]))
4
5  compare_bits = ParseFunction('iris_template_compare {IN} > {OUT}')
6
7  # 1. Generic version of AllPairs
8  results = AllPairs(compare_bits, bits_set, bits_set)
9  table   = Merge(results, 'table.txt')
10
11 # 2. Native version of AllPairs
12 results = AllPairs(compare_bits, bits_set, bits_set,'table.txt',
13             native=True, port=int(CurrentScript().arguments[2]))
```

Figure 4.6. Instruction Selection Benchmark Workflows

The top part of this code segment is the portion of code common to both workflows. Lines $7-9$ *contains the source for the generic version of All-Pairs. In this case, we perform an* `AllPairs` *and then a* `Merge` *to construct a table of results. For the optimized version, shown on Lines* $11-13$*, we simply set* `native` *to* `True` *and set the output file since the native tool automatically constructs a table for us.*

The results of these benchmarks are shown in Figure 4.7. From the graph, it is clear that the generic version scales linearly as the number of inputs increases (the total size of the inputs increases quadratically since all inputs are compared in a pair-wise fashion), while the optimized native version scales super-linearly relative to the size of the input dataset. The generic implementation is several

orders of magnitude slower because it must use files for intermediate storage and is unable to take advantage of specific execution engine environment features. For instance, the optimized *All-Pairs* tool uses the WorkQueue execution engine internally to enable low-latency work dispatching and takes advantage of multi-core systems by intelligently scheduling tasks to multiple cores. Moreover, the native tool aggregates intermediate outputs in memory and streams the outputs to a single file at the end of execution.

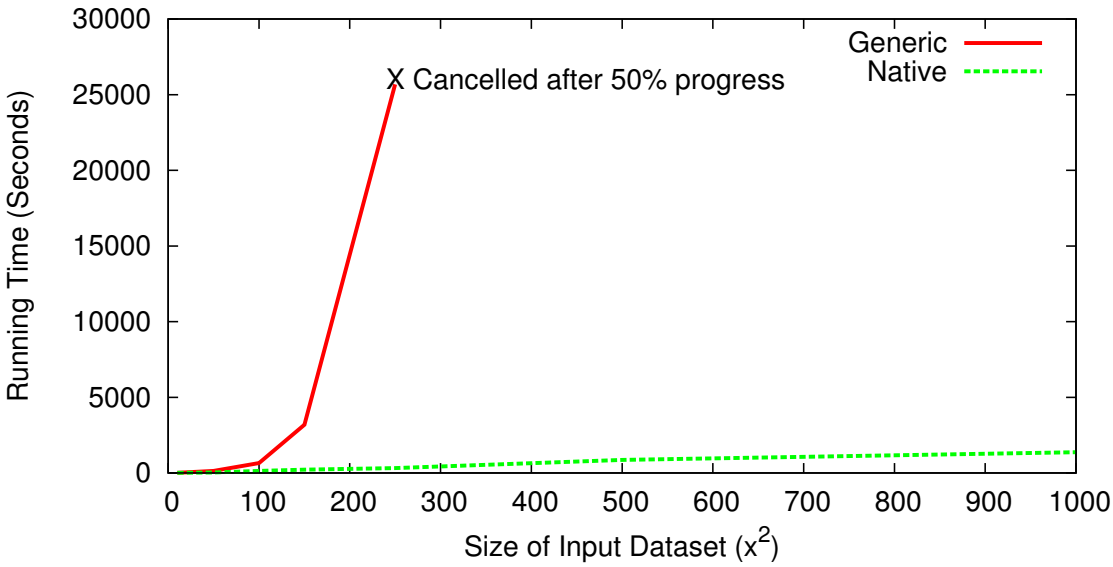


Figure 4.7. Instruction Selection Benchmark Results

In these benchmarks, the generic All-Pairs scales linearly with respect to the input dataset (which grows quadratically) and begins to produce unreasonable execution times around 250 input files. The native tool, on the other hand, performs super-linearly and continues to yield excellent performance even for the largest datasets.

As shown in the benchmark results demonstrated in Figure 4.7, there is a performance penalty associated with using the generic abstraction implementations. The Weaver programming model requires that input and output data must be stored as files, which greatly constrains the performance of certain types of workflows. In the benchmark shown, the generic version had to be stopped after around 250 input files since it was taking too long to complete (around 7 hours to reach 50% completion). In these cases, a specialized native tool easily outperforms the generic implementation because it is not constrained by the programming model. Fortunately, Weaver provides an *instruction selection* mechanism to take advantage of these optimized abstraction implementations, allowing for specialized versions to be used when available.

It is important to note that the availability of generic abstraction implementations is useful and necessary for the cases where a native implementation does not exist or does not match the semantics of the user's workflow. This allows users to still take advantage of a particular pattern of work, even if the tool is not available for their particular distributed computing platform.

Finally, the ability to choose between a generic implementation and an optimized one also makes Weaver flexible and attractive for exploring new abstractions. For instance, new patterns of execution can be quickly developed as a set of DAG relationships and tested on a variety of execution engines. If the new abstraction proves useful, then an optimized implementation can be developed and then plugged into Weaver seamlessly.

4.3 Hierarchical Workflows

Another method of optimizing a workflow is to allow the user to carefully utilize hierarchical `Nests` to structurally partition the workflow into smaller sub-workflows that can be executed concurrently. A high-level view of this transformation is shown in Figure 4.8 and is similar to *data partitioning* compiler technique where a sequentially iterated parallel loop is divided in a manner to minimize interprocessor communication and to software pipelining [65] in which iterations of a loop are continuously started at constant intervals.

When applied to scientific workflows, the idea behind utilizing these compiler techniques is to transform a flat monolithic DAG into a hierarchical organization of `Nests` such that each `Nest` represents to an independent sub-DAG that can be executed concurrently with its siblings. This hierarchical configuration, along with careful specification of execution platform parameters, allows for increase scalability and performance since multiple portions of the overall workflow can be executed in parallel.

The most straightforward way to construct a hierarchical workflow is to take a large dataset, split into evenly sized chunks, then treat each chunk as a smaller input dataset. To create the hierarchy, the user simply uses Python's `with` statement syntax with a `Nest` as explained in Section A.4, which creates a new child context with its own DAG and sandbox. Any `Abstractions` executed under this context will be attached to the created `Nest` rather than the parent.

This hierarchical arrangement should lead to scalability and performance improvements due to the following:

1. **Interleaving:** By partitioning the workflow into concurrent sub-DAGs and executing those sub-DAGs, more portions of the DAG can be active at once.

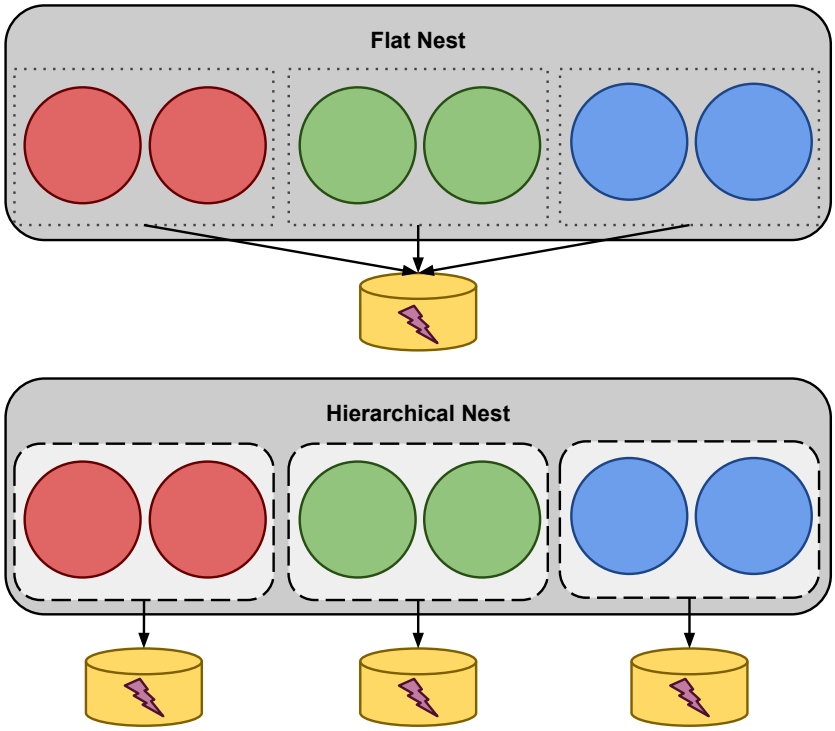


Figure 4.8. Hierarchical Workflows

In a hierarchical workflow, the end user uses multiple `Nests` *to partition the DAG into smaller nested sub-DAGs that can be executed concurrently. Each sub-DAG is encapsulated by a* `Nest` *and thus has its own sandbox and DAG separate from the parent* `Nest`*.*

Typically, a workflow engine such as Makeflow would execute as many independent tasks as possible, given its resource constraints. Some tasks, however, may tie up the workflow engine during data transfer or simply take a long time, preventing other tasks from being dispatched. By having multiple sub-DAGs with their own workflow engine instance (and thus master) portions of the DAG can perform I/O while other segments perform execution. In a single flat DAG, tasks would have to block while waiting for I/O to complete.

2. **Run-time Overhead:** Another performance advantage to partitioning the workflow is to overcome some of the performance limitations of the run-time system. For instance, parsing a workflow in Makeflow is a linear operation and thus becomes a small bottleneck in the startup of larger workflows. Moreoever, since Makeflow uses a fixed hash table representation and a linear scan algorithm for determining tasks to execute, processing and managing a larger workflow can take significantly more computational and memory resources that dividing the workflow into smaller, more manageable pieces.

In summary, hierarchical workflows allow for better resource utilization by enabling interleaving of I/O and execution and by reducing the amount of overhead incurred by the run-time system. By creating and using new `Nest` objects in conjunction with Python's `with` statement, end users can construct hierarchical workflows to take advantage of these possible performance enhancements.

To test the effectiveness of hierarchical workflows, I simplified the transcoding workflow currently used for the BXGrid project [23] into the benchmark workflows shown in Figure 4.9. In these benchmark workflows, I queried seven different types of biometrics data (`ABS`, `CR2`, `image`, `POE`, `RAW`, `iris`, and `video`) to produce one workflow that transcodes a certain number of files ($1,000$ in these tests) of those types in a single flat workflow and a second workflow that performed the same transcoding using a hierarchical set of `Nests` instead.

For these benchmarks, I took the workflows in Figure 4.9 and created two sets: **(1)** the first set of workflows involved transcoding only *Small* files (i.e. excluding `video` and `POE` files) using a flat and a hierarchical workflow, **(2)** a second set of *Big* workflows that included all seven types of files mentioned previously using both flat and hierarchical workflows. To test these configurations I used Makeflow's

```
# 0. Transcoding datasets
bxgrid   = MySQLDataset(HOST, 'biometrics', 'files')
datasets = [
    ('abs', 'convert_abs_to_gif_animation {IN} 512 384 {OUT}',
     Query(bxgrid, bxgrid.c.fstate == 'ok',
                   bxgrid.c.extension | ('gz', 'abs.gz'), limit=LIMIT),
     '{BASE_WOEXT}.gif'),
    ('cr2', 'convert_cr2_to_jpg {IN} 512 384 {OUT}',
     Query(bxgrid, bxgrid.c.fstate == 'ok',
                   bxgrid.c.extension == 'cr2', limit=LIMIT),
     '{BASE_WOEXT}.jpg'),
    ('image', 'convert_image_to_jpg {IN} 512 384 {OUT}',
     Query(bxgrid, bxgrid.c.fstate == 'ok',
                   bxgrid.c.extension | ('JPG', 'ppm'), limit=LIMIT),
     '{BASE_WOEXT}.jpg'),
    ('POE', 'convert_POE_to_gif_animation {IN} 512 384 {OUT}',
     Query(bxgrid, bxgrid.c.fstate == 'ok',
                   bxgrid.c.extension == 'b3d', limit=LIMIT),
     '{BASE_WOEXT}.gif'),
    ('raw', 'convert_raw_to_jpg {IN} 512 384 {OUT}',
     Query(bxgrid, bxgrid.c.fstate == 'ok',
                   bxgrid.c.extension == 'NEF', limit=LIMIT),
     '{BASE_WOEXT}.jpg'),
    ('tiff', 'convert_iris_to_template {IN} {OUT}',
     Query(bxgrid, bxgrid.c.fstate == 'ok',
                   bxgrid.c.extension == 'tiff', limit=LIMIT),
     '{BASE_WOEXT}.bit'),
    ('video', 'convert_video_to_gif_animation {IN} 512 384 {OUT}',
     Query(bxgrid, bxgrid.c.fstate == 'ok',
                   bxgrid.c.extension | ('avi', 'mp4', 'MPG', 'ts'),
                   limit=LIMIT),
     '{BASE_WOEXT}.gif'),
]
# 1. Flat Transcoding Workflow
for type, convert, files, output in datasets:
    Map(convert, files, output)

# 2. Hierarchical Transcoding Workflow
for type, convert, files, output in datasets:
    with ParrotNest(type):
        Map(convert, files, output)
```

Figure 4.9. Transcode Benchmark Workflows

This is a simplified version of the transcoding workflow used for the BXGrid website. The first workflow transcodes all the different files in a single DAG while the second one performs the transcoding using hierarchical `Nests`.

WorkQueue batch system with a pool of 100 workers and executed these workflows multiple times. For the hierarchical workflows, I had all of the sub-DAGs run as local jobs on the parent node.

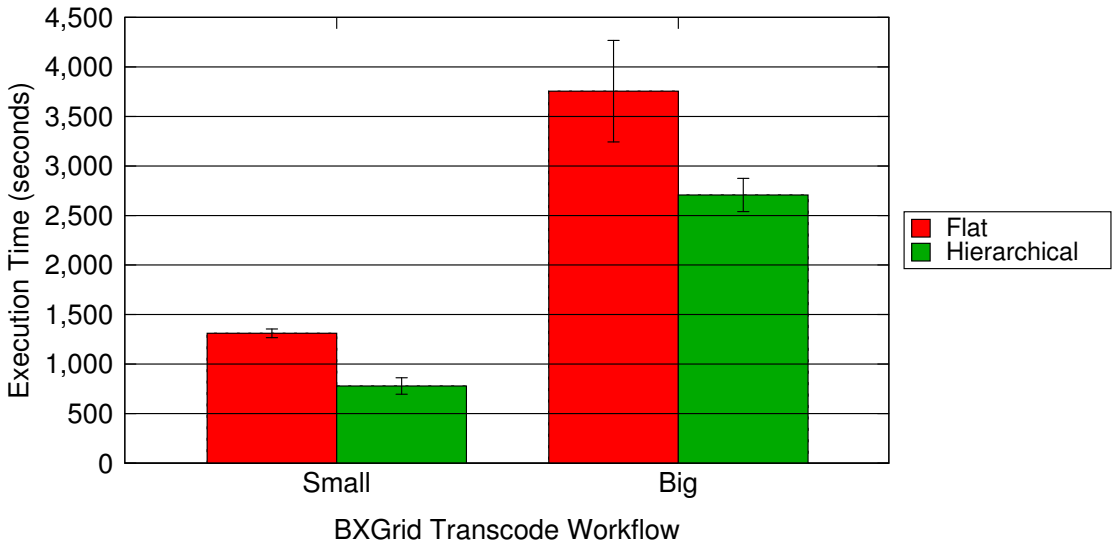


Figure 4.10. Transcode Benchmark Execution Times

In this graph, the Small workflow used only smaller files (i.e. without `video` *and* `POE` *data), while the Big workflow included all seven types of data. As can been seen, hierarchical workflows performed better than the flat monolithic workflow. This difference in performance between flat and hierarchical, however, was larger for the Small workflow than the Big workflow.*

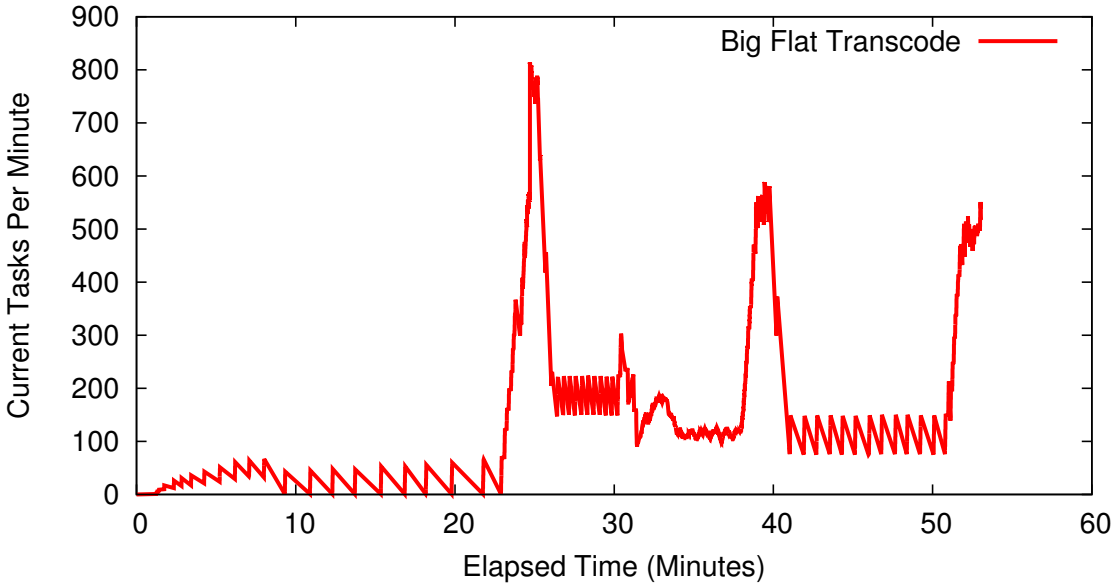
The results of these benchmarks are shown in Figure 4.10. As can be seen, the hierarchical workflows performed much better than the flat monolithic workflows in transcoding the biometric data. In the case of the *Small* workflow, the hierarchical version cuts the execution time nearly in half. For the *Big* workflow,

the hierarchical version yielded a significant performance increase but not nearly as much as it did for the *Small* workflow.

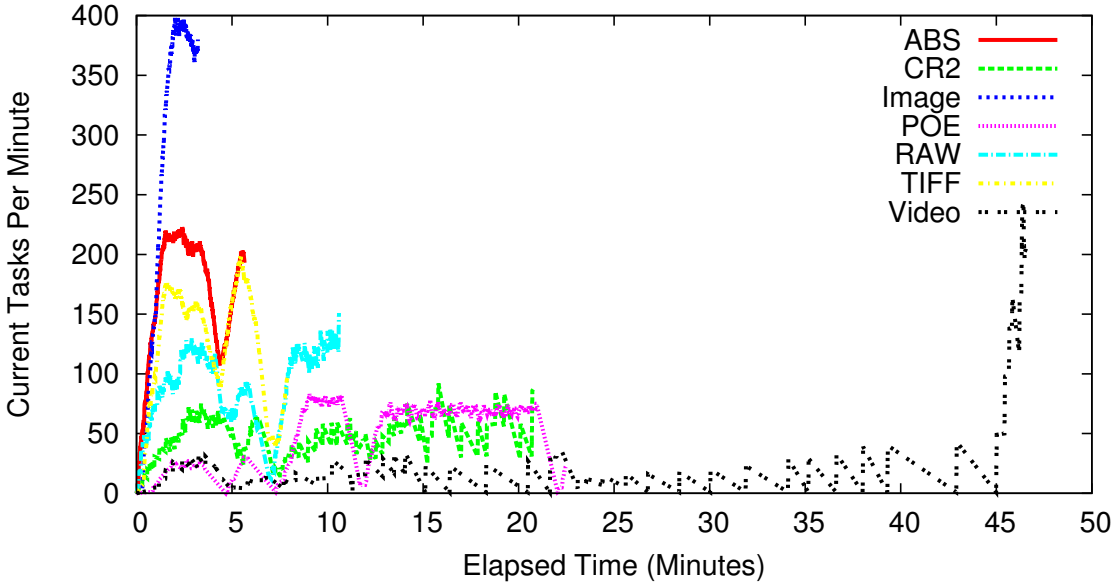
The cause of these performance increases is made clear in Figure 4.11. These two charts show the *Current Tasks/Minute* rate (how many tasks have completed in the last minute) of a *Big* flat transcoding workflow and a *Big* hierarchical transcoding workflow. As demonstrated in the top graph, there are many points in the graph where the completion rate drops to zero. This is because the workflow engine, Makeflow is so busy transferring data to workers that it cannot receive output data from finished workers and cannot dispatch new tasks. The reason for this long period of I/O is that some of the video files were quite large (i.e. multiple gigabytes) and thus forced a stall in the pipeline. The second graph supports this analysis as even in the hierarchical version, the video sub-DAG also experiences dips to zero in the completion rate, which is evidence of large file transfers.

The hierarchical workflow was able to mitigate or workaround this huge I/O bottleneck because it while the video sub-DAG blocked on transferring files, the other sub-DAGs were able to continue processing. As predicted above, hierarchical workflows enabled interleaving of I/O and execution across a wider portion of the graph and thus increased the performance of the workflow.

There is a limit to this performance increase, however, as shown in Figure 4.10. In running the benchmarks, all the masters of the hierarchical workflow ran on a single machine. This means that there were periods of time during the execution of the workflow that the network bandwidth was saturated. This explains why the *Big* workflow did not see as much of a performance increase as the *Small* one: in the *Big* workflow the video files dominated the outgoing network bandwidth and served as a bottleneck on how quickly tasks could be dispatched. The solution



(a) Big Flat Transcode



(b) Big Hierarchical Transcode

Figure 4.11. Transcode Benchmark Task Rates Over Time

As can be seen this first graph, the flat version has lots of dips in task per minute completion rate. This means that there were many moments during the execution of the workflow where the workflow engine was busy transferring data out instead of receiving data and dispatching new tasks. Contrast this with the bottom graph, where only the video line shows huge dips to zero, while all the other file types are able to maintain a positive completion rate.

to this is to distribute the multiple masters in a hierarchical workflow to different machines in order to spread out the data transfers. This was beyond the scope of my immediate research, however, so I leave it as possible future work.

The overall takeaway from these results is that using `Nests` to construct hierarchical workflows can yield significant performance increases due to interleaving of tasks across multiple sub-DAGs and minimizing of run-time overhead. In particular, workflows with large datasets that can be divided into smaller independent chunks are well-suited for this technique. For workflows composed of a long sequence of inter-dependent phases, however, hierarchical workflows do not offer much of an advantage and may in fact hinder it as each stage of the workflow must block for the entire sub-DAG to complete before dispatching any of the tasks of the subsequent sub-DAG.

4.4 Inlining Tasks

The final optimization method investigated in this dissertation is inlining tasks. When this technique is enabled by the user, groups of similar tasks are aggregated into super-tasks which consist of sub-DAGs that perform the collected tasks as shown in Figure 4.12. This optimization method is similar to inlining functions [30] and unrolling loops [12, 38] in conventional programming languages. The principle behind this technique is to minimize the dispatch latency by reducing the overhead involved in executing the desired operations. By aggregating similar tasks together into super-tasks, we reduce the amount of tasks the top-level workflow manager has to oversee. This technique has been used in the past in DAG-based workflow system such as Pegasus, which refers to this particular method as "level clustering" [63].

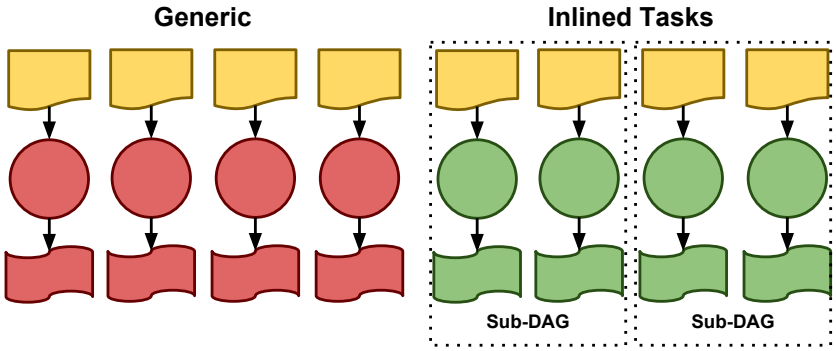


Figure 4.12. Inlined Tasks

When using the Weaver compiler, a user may either perform this optimization globally by passing the `-t <group_size>` parameter or by specifying a value to an `Abstraction's group` keyword argument. In either case, the `group` size is used to divide the schedule tasks associated with each `Abstraction` into evenly sized groups. That is, after each `Abstraction` is compiled and a set of tasks is generated, the compiler checks to see if the global `group_size` parameter is set or if the `Abstraction's group` attribute is set. If so, the compiler will take the generated set of tasks and aggregate them into sub-groups based on the specified parameter and then schedule these meta-tasks.

To test the effectiveness of inlining tasks, I used the workflow shown in Figure 4.13. In this workflow, I create a series of tasks that simply sleep for a random amount of time and then create the specified output file. The total amount of tasks to be executed is passed as an argument to the workflow during compilation. In my benchmarking I used $1,000$ as my `ITERATIONS` amount and executed this workflow using Makeflow's local, Condor, and WorkQueue batch systems on a 16-core machine. Most importantly, I ran the workflow with varying global

```
import random
random.seed(1)

ITERATIONS = int(CurrentScript().arguments[0])
Iterate('sleep {ARG}; install -D /dev/null {OUT}',
    [random.randint(0, 5) for i in range(ITERATIONS)], '{stash}')
```

Figure 4.13. Inlined Tasks Benchmark Workflow

This benchmark simply performs N tasks where each task involves sleeping a random amount of time and then creating a file and where N is the number of tasks to execute as specified by `ITERATIONS`.

`group_sizes` (e.g. 1, 2, 4, 8, 16) to determine the affect of inlining tasks on these different distributed execution platforms.

The results of these benchmarks are shown in Figure 4.14. On all three distributed execution platforms, as we increase the amount of inlining we do (i.e. increase the `group_size`), we also increase the performance of the workflow. In other words, the more we inline tasks, the more speedup relative to not inlining we accomplish. Condor, in particular, received the biggest performance boost, which is not a surprise since it has the largest dispatch latency and most variability. This volatility also explains why for larger `group_sizes` we experience a reduction in speedup relative to previous sizes; it is most likely that my Condor user priority prevented tasks from executing in a timely manner towards the end of the benchmarking.

While the workflows on Condor experienced significant speedups, the workflows on local and WorkQueue demonstrated more modest running time improvements. This could be simply due to the fact that both of these batch systems are already low latency execution platforms, so the advantage of aggregating similar tasks is lessened on these faster systems. It is interesting to note, however, that

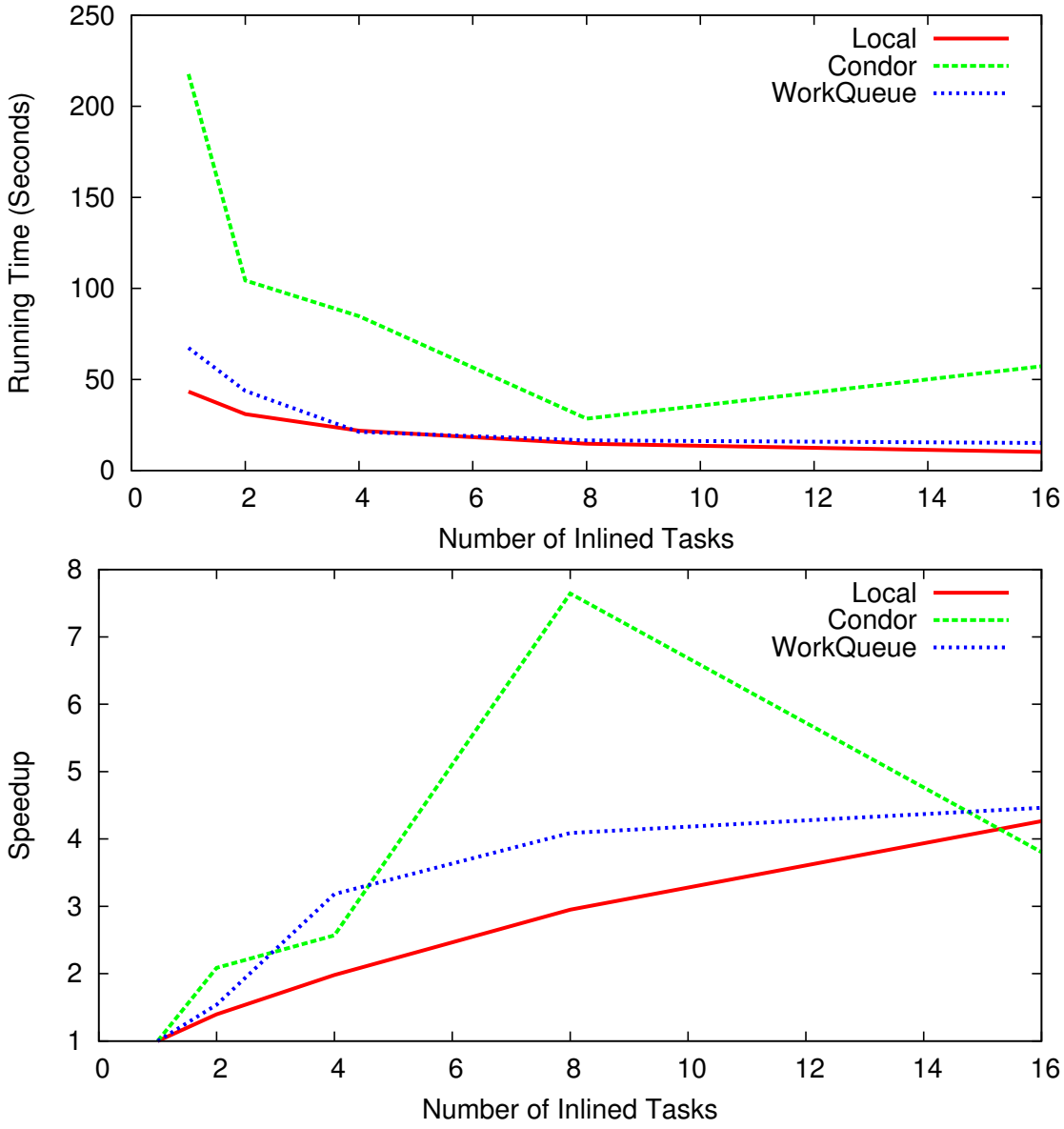


Figure 4.14. Inlined Tasks Benchmark Results

As the number of inlined tasks increases, we see performance increases across all three distributed execution engines. Unsurprisingly, the workflows on Condor experienced the most speedup, as that execution platform has worst and most variable dispatch latency.

inlining brought the running time of the WorkQueue workflows down to the same level as the local workflows. That is, due to task inlining, executing the workflows using WorkQueue was just as fast as using the local batch system. Compared to the local system, WorkQueue experienced greater speedups, but since it was slower than the local system to begin with, it never surpassed it in terms of actual execution time.

Altogether, these four optimizations manifest the effectiveness of applying traditional compiler techniques to distributed workflow generation. The `Stash` structure addresses the problem of intermediate files by providing the user a convenient method of automatically managing the workflow namespace in a manner similar to register allocation. The Weaver compiler allows the end user to perform instruction selection by supporting the `native` keyword argument to abstractions which will replace the generic DAG implementation of the pattern with a native optimized tool. Using `Nests`, users can partition their workflows into hierarchical workflows that improve the scalability and performance of their applications through increasing interleaving of tasks and minimizing run-time overhead. Finally, inlining tasks allows users to minimize the dispatch latency of the distributed execution engine in a manner similar to how compilers perform loop unrolling or function inlining. As the results presented in this chapter show, all of these compiler techniques can greatly improve the performance of the generated workflow when used appropriately.

CHAPTER 5

LINKING WORKFLOWS

While the compiler is the central component of a programming toolchain, many software development suites include a variety of auxiliary utilities. A critical utility in a conventional toolchain is the *linker*, which is a special program that takes one or more objects produced by a compiler and combines them into a single unit (e.g. executable, shared library, etc.). In the traditional GCC toolchain [106], the `LD` linker combines various object and library archives, relocates the application data, and modifies symbol references for proper run-time resolution. During this linking process, `LD` may also perform various optimizations such as removing unnecessary instructions or modifying calling procedures [10].

In the context of distributed workflows, we are mainly concerned with the packaging and reference modification capabilities of a linker. As noted previously, distributed workflows are generally *ensembles* that consist of multiple executables and libraries working together to form a single *meta*-application. Within this programming environment we face two main problems: **(1)** Applications with external dependencies and **(2)** Workflows with many components and static workspaces.

To address the first problem, my solution is to combine all the relevant components of an application into a self-extracting executable archive. For normal binaries, this involves packaging the executable along with its library dependencies. For applications such as Python or Matlab scripts, this involves archiving

the script and the language interpreter and run-time system in order to produce a self-contained executable. The general principle is to link in any external dependencies into a single portable unit for reliable distribution and execution.

The solution to the second problem is to modify the paths and references in the workflow's directed acyclic graph in order to enclose the namespace of application. This normally involves converting absolute paths to locations internal to the workflow's workspace. To increase portability even further, input files and executables may also be copied to the workspace. Because the references of the workflow have been modified to only point to internal files, the linked workflow is more portable and less reliant on the particular environment in which it is created or executed.

The remainder of this chapter describes how these linking principles are utilized in two utilities provided by the compiler toolchain. The first is an application linker that allows users to create self-contained applications for a wide variety of applications including non-binary executables such as Python and Matlab scripts. The second is a workflow linker that modifies the internal references to make the workflow less dependent on the environment in which it was created.

5.1 Application Linker

It is common for individual applications to depend on external files such as shared libraries or run-time data. On a single machine, the existence of multiple external dependencies is not a significant problem. However, in a distributed environment, the application is most likely to be transfered to a remote machine where its requirements may not be sufficiently satisfied. As such, applications with external dependencies pose a significant problem for distributed workflows.

For executables that depend on shared libraries, the most straightforward solution is to simply *statically* link external libraries into the target executable. This avoids the need to dynamically locate and load the libraries at run-time (i.e. the libraries are instead embedded in the executable). A potential downside to this solution is that static linking increases both the size of the program (thus transfer costs) and the amount of memory required for execution [33]. In the context of a distributed workflow, however, this resource trade-off is reasonable as it allows the user to avoid the complexity of managing the external dependencies of an application across multiple machines.

Unfortunately, it is not always possible for end-users to create a static executable without access to the original source or object files. Moreover, files such as configuration settings and other run-time data cannot be linked into the executable using the traditional `LD` utility. Applications which invoke other executables (e.g. shell scripts) also cannot be statically linked but have external dependencies that must be managed. For these situations, an *application linker* is required to package the external dependencies of a program into a single execution unit for manageable distribution.

Although the compiler supports specifying a `Function`'s dependencies, it is often desirable to execute these individual applications independently (i.e. outside the context of the workflow). Moreover, some programs require data or configuration that are difficult or cumbersome to specify in Weaver. To solve this problem, the toolchain includes an *application linker* named **Starch** [114] to create *standalone application archives* (*SAA*). These packaged executables are self-extracting archives that contain all of the dependencies, configuration, and environment settings necessary to execute the particular application.

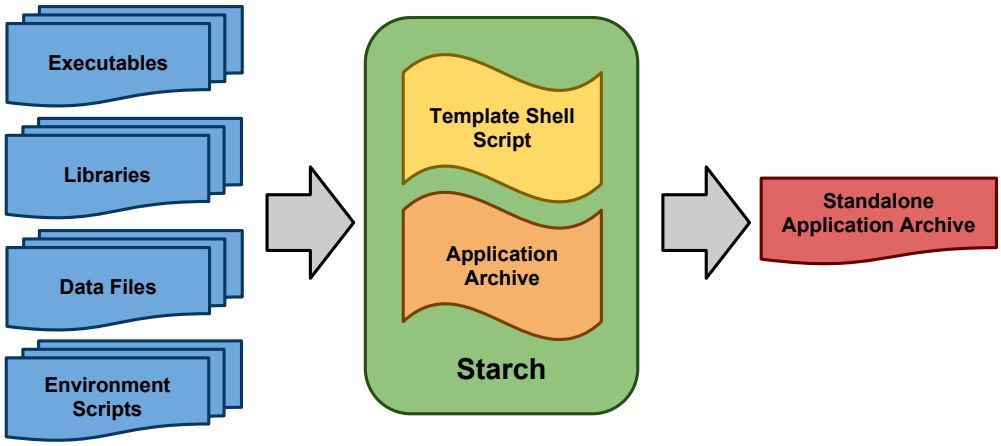


Figure 5.1. Starch (STandalone application ARCHiver)

To create a standalone application archive, the user gives Starch a list of executables, libraries, data files, and environmental scripts to package. The resulting artifact is a self-extracting executable that consists of a shell script and a tarball embedded at the end of the script.

As illustrated in Figure 5.1, to create a *SAA* using Starch, the user specifies a list of executables and libraries to include in the target execution image along with the appropriate shell command to run when the *SAA* is executed by the user. By default, Starch will automatically search for any dynamically linked libraries required by the executables specified and embed those in the archive. If any special input data files are required by the application, they may also be specified to be included in the *SAA*. Additionally, special environmental variables and other run-time configuration options can be stored in the package through the use of user-defined environment scripts that will be imported before the *SAA*'s command is executed.

Once all of the necessary options are specified, the executables, libraries, and environment scripts are archived and compressed as a standard Unix tarball. This

tarball is then appended to Starch's template shell script to generate a standalone application archive. When the *SAA* is executed the wrapper shell script will automatically extract the embedded archive, configure the environment, and run the user specified shell command. To utilize Starch generated application packages in their workflows, users simply replace the normal executables with the constructed SAA's in their workflow specifications.

TABLE 5.1

STARCH CREATION AND EXTRACTION BENCHMARK

Program	**LIBS**	**EXE Size**	**SAA Size**	**Creat. Time**	**Extract. Time**
convert	27	24.83 KB	7.13 MB	3.37 ± 0.35	1.39 ± 0.17
ffmpeg	10	6.44 MB	4.46 MB	2.55 ± 0.21	0.95 ± 0.11

Table 5.1 provides the results of creating and extracting the `convert` and `ffmpeg` applications using Starch. The first application, `convert`, was dynamically linked and had many library dependencies (27). This bloated the *SAA* size to $7.13MB$ from the $24.83KB$ of the original executable. The second program, `ffmpeg`, had fewer dependencies and actually decreased in size, with the *SAA* at $4.46MB$ and the original command at $6.44MB$. This is most likely due to compressing long text strings embedded in the executable and the libraries. Unsurprisingly, the creation and extraction times for `convert` was greater than

`ffmpeg`, since the former was larger in file size and had more components. Because *SAAs* are extracted before execution, the extraction times in Table 5.1 reveal that using *SAAs* for short tasks could severely degrade performance. That is, if the execution time if the task is less than the extraction time of the *SAA* then the overhead incurred in using the *SAA* will adversely affect the running time of the workflow.

```
# 1. Normal Convert
convert = ParseFunction('convert {IN} {OUT}')

jpgs = Glob('{0}/*.jpg'.format(CurrentScript().arguments[0]))
Map(convert, jpgs, '{basename}.png')

# 2. Starched Convert
convert_sfx = ParseFunction('convert.sfx {IN} {OUT}',
                environment={'SFX_UNIQUE': 1})

jpgs = Glob('{0}/*.jpg'.format(CurrentScript().arguments[0]))
Map(convert_sfx, jpgs, '{basename}.png')

# 3. Starched Convert with keep option
convert_sfx = ParseFunction('convert.sfx {IN} {OUT}',
                environment={'SFX_KEEP': 1})

jpgs = Glob('{0}/*.jpg'.format(CurrentScript().arguments[0]))
Map(convert_sfx, jpgs, '{basename}.png')
```

Figure 5.2. Starch Benchmark Workflows

In all of the benchmarks we convert a set of JPG images to PNG format. The first benchmark workflow uses the normal `convert` *executable, while the second one uses a* `Starched` *version of the application. The third one uses a* `Starched` *convert but with the* `SFX_KEEP` *flag that lets the SAA to use a previously extracted version of the application.*

To test the impact of using *SAA* instead of normal executables, I constructed three simple workflows as shown in Figure 5.2 and executed them using Makeflow's `local` batch system on a 16-core machine. The goal of all three benchmarks is to convert a set of JPG images to PNG format. The first benchmark workflow uses the system installed `convert` executable, while the second one uses a *SAA* linked by Starch that is extracted before each invocation. The last workflow also uses the convert*SAA* but with the `SFX_KEEP` flag enabled. When set to a positive value, the `SFX_KEEP` environment variable allows the *SAA* to use a previously extracted version of the application, and thus avoid the cost of extracting the archive.

The results of these benchmarks are shown in Figure 5.3. As can be seen in the chart, using the normal `convert` executable was almost twice as fast as using the *SAA* in the workflow where we had extract the archive every time we executed the application. By using the `SFX_KEEP` flag in the third workflow, we are able to ameliorate the costs associated with using a *SAA* by only extracting the archive once (i.e. the first time it is executed). Using this technique, the third workflow is much closer to the first one in terms of execution time. The small difference in performance can be attributed to the remaining overhead incurred when using a *SAA*: checking for the extracted archive and trampolining the shell command.

Although the *application linker* is conceptually simple, Starch is a useful utility in the compiler toolchain that enables users to package complex executables for use in distributed workflows. It is currently being used by collaborators to simplify and package a variety of bioinformatics workflows [67, 114] which incorporate many executables and libraries that need to work across multiple distributed systems.

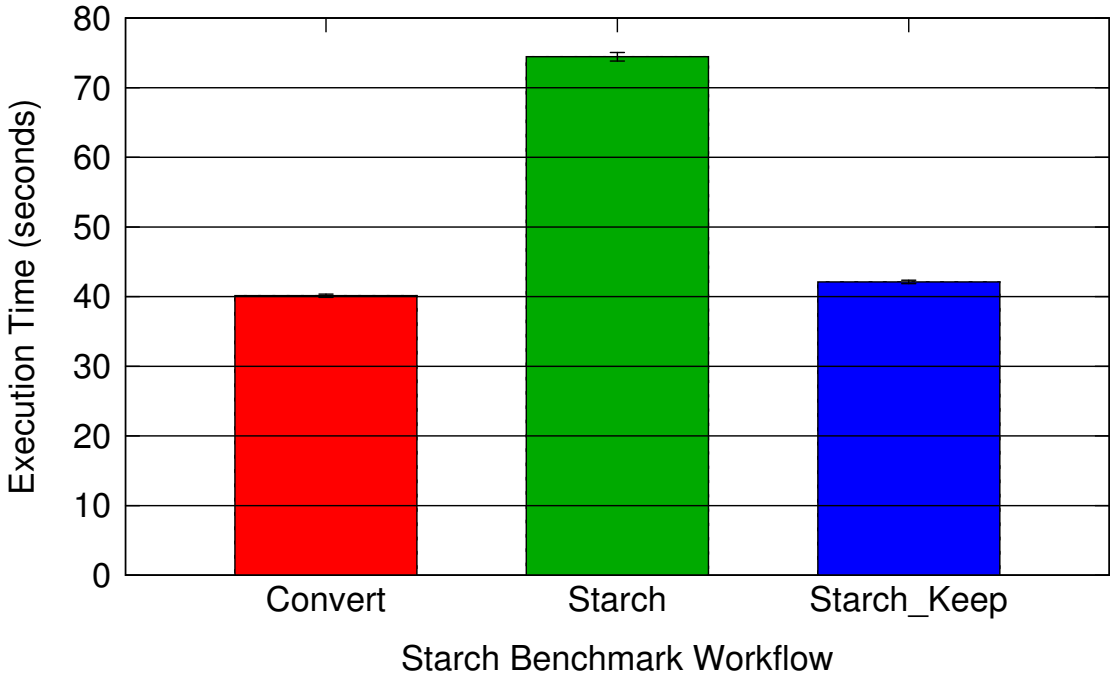


Figure 5.3. Starch Benchmark Workflow Execution Times

Using SAAs can incur significant execution overheads if the execution time of the application is dwarfed by the time it takes to extract the archive. One way to mitigate this overhead is by setting the `SFX_KEEP` *to a positive value, which will allow the SAA to skip extracting the archive it the target directory already exists. This graph shows that this technique can bring the execution time of a workflow using SAAs down to close proximity to a workflow using the normal executables.*

5.2 Workflow Linker

In addition to the Starch *application linker*, the toolchain also includes a *workflow linker* called `makeflow_link`. Unlike Starch, which mainly deals with packaging a single application, the `makeflow_link` utility operates on the whole workflow and primarily performs reference or symbol manipulation on the DAG, which is another common responsibility for traditional linkers such as `LD`. The `makeflow_link` tool supports the following linking actions:

1. **Copying:** `makeflow_link` can be used copy available input and executable files to the workflow's workspace. Copying these files to the workspace would allow users to simply archive up the folder containing the workflow to form a snapshot of their workflow or to share with collaborators.

2. **Symlinking:** Instead of copying input or executable files, `makeflow_link` can be instructed to symlink these files to the workflow's workspace. This can be useful for distributed execution platforms that expect all files to be inside the workspace. If snapshotting or archiving is not desired, symlinking allows the user to meet the execution platform's requirements without the cost of copying many files to the local sandbox.

3. **Starching:** Executables can automatically be converted into standalone application archives by having `makeflow_link` run Starch on each of these applications. Using this option in conjunction with copying input files should produce a portable workflow that can be re-located to other systems with relative ease.

The linker also supports options for specifying whether input, output, `Stash`, or executable files should use relative or absolute paths. By default, Weaver generates workflow DAGs that contain absolute paths. While most of the distributed execution engines supported by Makeflow can handle absolute paths, there are some situations where relative paths are required or preferred. `makeflow_link` can systematically convert these paths for the user and ensure that all rules are updated properly. Since `Stash` and executable files can be either input or output files, any options associated with these types of files will take precedence of the options for input or output files. Note that copying, symlinking, or Starching a file implies the use of a relative path for that particular file.

```
Usage: makeflow_link [options] <DAG> ...

Options:
  -h, --help           show this help message and exit

  General Options:
    -d COPY_DIRECTORY  Copy or symlink input files to this sub-directory.
    -r                 Recursively link nested DAGs.
    -v                 Report progress information.

  Backup Options:
    -b                 Backup existing DAG before overwriting.
    -s SUFFIX          Use this as suffix for DAG backup name.

  Linking Options:
    Input, Output, Stash, and Executable files may be set a combination of
    the following linking options: (1) copy: copy input or executable
    files to sandbox, (2) symlink: symlink input or executable files to
    sandbox, (3) starch: starch executables, (4) absolute: use absolute
    file path in rules, (5) relative: use relative file path in rules. The
    Stash and Executable settings take precedence over the Input and
    Output settings.

    -I options         Set linker options for input files.
    -O options         Set linker options for output files.
    -S options         Set linker options for stash files.
    -X options         Set linker options for executable files.
```

Figure 5.4. `makeflow_link` Command Line Options

This linker allows the user to specify various linker options for input, output, `Stash`*, and executable files. Options for the latter two take precedence over the first two. The utility over-writes the DAG, but the user can request the original is backed up. Additionally, the command can detect nested sub-workflows and recursively link those as well.*

As an additional feature, if the workflow contains sub-workflows as in the case of hierarchical workflows or when using inlined tasks, the `makeflow_link` can recursively link the all of the nested workflows to ensure that the entire collection of workflows is consistently linked. To ensure that inter-DAG dependencies are updated properly, the DAG is traversed recursively in a depth-first manner. This

means that a DAGs children are linked before the parent's DAG is processed by the linker.

The *workflow linker* works by reading in the Makeflow DAG and parsing it using the `cctools.makeflow.dag` module from the `python-cctools` library. This module provides a recursive-descent parser that translates a Makeflow DAG into a Python data structure. Once the DAG is read and parsed into memory, the `makeflow_link` traverses the graph starting with the children first to ensure that any path modifications performed in the sub-workflows are propagated to the parent workflow.

To perform the actual linking, the `makeflow_link` application looks at each node in the DAG and determines if any of the files associated with the node needs to be modified according to the command line arguments passed by the user. As mentioned above, linker options for `Stash` and executable files take precedence over the options for input and output files. When each file in node is examined, all of these linker options are analyzed to determine whether or not to copy, symlink, or starch a file and whether or not to use relative or absolute paths. If no options is specified for that particular type of file, then no action is performed. This means that if a file is in absolute format, it will remain in that format unless the user explicitly requests for it to be converted to relative form. During this checking, if any files are renamed then the node's command is updated to reflect these changes using Python's string replacement methods. Once the whole DAG has been parsed and linked, then it is written to the original DAG file. For convenience, the linker can be told to backup the original file before writing the new DAG.

Figure 5.4 displays the command line options to `makeflow_link`. Suppose a user wanted to modify a workflow such that all the input files were copied to

the sandbox, all executables are symlinked and relative, and all output files used relative paths. From my experience, it is quite common for the user to put all of the input data into the workflow's sandbox, keep the executables where they are, and capture all of the output data in the local workspace. To accomplish this configuration, the user would use the following invocation of `makeflow_link`:

```
makeflow_link -I copy -O relative -X relative,symlink Makeflow
```

The result of this command is demonstrated in Figure 5.5. The top portion of the listing shows the original DAG before linking was performed. As can be seen, all of the paths are in absolute format because that is what the compiler generates. The bottom half of the listing displays the resulting DAG after `makeflow_link` has transformed the graph such that all inputs are copied to the workflow's workspace, all executables are symlinked and use relative paths, and all output files are referred to using relative paths. Note that for executables such as `touch` are prefixed with "./". This is because the Makeflow commands are treated as *shell commands* and if the current directory is not in the `PATH` environmental variable, then the command will fail without the prefix.

```
# Before linking
/tmp/pbui-weaver-tests/iterate/0: /usr/bin/touch
        /usr/bin/touch /tmp/pbui-weaver-tests/iterate/0
/tmp/pbui-weaver-tests/iterate/1: /usr/bin/touch
        /usr/bin/touch /tmp/pbui-weaver-tests/iterate/1
/tmp/pbui-weaver-tests/iterate/2: /usr/bin/touch
        /usr/bin/touch /tmp/pbui-weaver-tests/iterate/2
/tmp/pbui-weaver-tests/iterate/0.stat: /tmp/pbui-weaver-tests/iterate/0 /usr/bin/stat
        /usr/bin/stat /tmp/pbui-weaver-tests/iterate/0 > /tmp/pbui-weaver-tests/iterate/0.stat
/tmp/pbui-weaver-tests/iterate/1.stat: /tmp/pbui-weaver-tests/iterate/1 /usr/bin/stat
        /usr/bin/stat /tmp/pbui-weaver-tests/iterate/1 > /tmp/pbui-weaver-tests/iterate/1.stat
/tmp/pbui-weaver-tests/iterate/2.stat: /tmp/pbui-weaver-tests/iterate/2 /usr/bin/stat
        /usr/bin/stat /tmp/pbui-weaver-tests/iterate/2 > /tmp/pbui-weaver-tests/iterate/2.stat

# After linking
0: ./touch
        ./touch 0
1: ./touch
        ./touch 1
2: ./touch
        ./touch 2
0.stat: ./stat 0
        ./stat 0 > 0.stat
1.stat: ./stat 1
        ./stat 1 > 1.stat
2.stat: ./stat 2
        ./stat 2 > 2.stat
```

Figure 5.5. Result of Linking with `makeflow_link` (1)

In this example the original DAG is transformed by copying all the inputs to the workflow's workspace, symlinking the executables, and using for output files and executables.

Now suppose that the user wishes to modify a workflow such that all inputs are left alone, but outputs are relative, and executables are converted into standalone application archives. This type of situation may occur if the input data is large and thus is quite prohibitive to copy to a local sandbox, but the user still wants to capture the outputs to the workflow's workspace and to create *SAAs* for increased portability. To achieve this setup, the following command may be used:

```
makeflow_link -O relative -X starch Makeflow
```

Figure 5.6 demonstrates the output of such a command. In this example, `makeflow_link` automatically packages the `convert` and `stat` executables into standalone application archives and all of the output and executable paths are converted to relative paths. The input files, however, continue to use absolute paths since we did not specify any linker options for input data.

As manifested in these examples, the `makeflow_link` utility is a powerful tool for systematically manipulating a generated workflow DAG. It can be used to make the workflow more portable by copying files, archiving executables, or converting paths to relative format. Moreover, the utility also sanitizes the DAG by removing duplicates from input and output file lists and compacts the DAG by stripping unnecessary whitespace.

Because packaging applications and modifying the generated DAG are common and desirable operations, the toolchain includes two linker applications. The first is Starch, which serves as an *application linker* that allows for convenient packaging of programs, libraries, data files, and environment configurations. The second tool is the `makeflow_link` *workflow linker* that provides the ability to systematically modify and the entire DAG to make it more portable or amendable to the distributed execution platform.

```
# Before linking
/tmp/pbui-weaver-tests/map/xterm-color_32x32.jpg: /usr/share/pixmaps/xterm-color_32x32.xpm /usr/bin/convert
        /usr/bin/convert /usr/share/pixmaps/xterm-color_32x32.xpm /tmp/pbui-weaver-tests/map/xterm-color_32x32.jpg
/tmp/pbui-weaver-tests/map/xterm_32x32.jpg: /usr/share/pixmaps/xterm_32x32.xpm /usr/bin/convert
        /usr/bin/convert /usr/share/pixmaps/xterm_32x32.xpm /tmp/pbui-weaver-tests/map/xterm_32x32.jpg
/tmp/pbui-weaver-tests/map/parcellite.jpg: /usr/share/pixmaps/parcellite.xpm /usr/bin/convert
        /usr/bin/convert /usr/share/pixmaps/parcellite.xpm /tmp/pbui-weaver-tests/map/parcellite.jpg
/tmp/pbui-weaver-tests/map/nest.py.stat: /home/pbui/src/research/weaver/weaver/nest.py /usr/bin/stat
        /usr/bin/stat /home/pbui/src/research/weaver/weaver/nest.py > /tmp/pbui-weaver-tests/map/nest.py.stat
/tmp/pbui-weaver-tests/map/data.py.stat: /home/pbui/src/research/weaver/weaver/data.py /usr/bin/stat
        /usr/bin/stat /home/pbui/src/research/weaver/weaver/data.py > /tmp/pbui-weaver-tests/map/data.py.stat
/tmp/pbui-weaver-tests/map/function.py.stat: /home/pbui/src/research/weaver/weaver/function.py /usr/bin/stat
        /usr/bin/stat /home/pbui/src/research/weaver/weaver/function.py > /tmp/pbui-weaver-tests/map/function.py.stat

# After linking
xterm-color_32x32.jpg: /usr/share/pixmaps/xterm-color_32x32.xpm ./convert.sfx
        ./convert.sfx /usr/share/pixmaps/xterm-color_32x32.xpm xterm-color_32x32.jpg
xterm_32x32.jpg: /usr/share/pixmaps/xterm_32x32.xpm ./convert.sfx
        ./convert.sfx /usr/share/pixmaps/xterm_32x32.xpm xterm_32x32.jpg
parcellite.jpg: ./convert.sfx /usr/share/pixmaps/parcellite.xpm
        ./convert.sfx /usr/share/pixmaps/parcellite.xpm parcellite.jpg
nest.py.stat: /home/pbui/src/research/weaver/weaver/nest.py ./stat.sfx
        ./stat.sfx /home/pbui/src/research/weaver/weaver/nest.py > nest.py.stat
data.py.stat: ./stat.sfx /home/pbui/src/research/weaver/weaver/data.py
        ./stat.sfx /home/pbui/src/research/weaver/weaver/data.py > data.py.stat
function.py.stat: ./stat.sfx /home/pbui/src/research/weaver/weaver/function.py
        ./stat.sfx /home/pbui/src/research/weaver/weaver/function.py > function.py.stat
```

Figure 5.6. Result of Linking with `makeflow_link` (2)

In this example the original DAG is transformed by converting all of the executables to standalone application archives and using relative paths for the output files and executables.

CHAPTER 6

PROFILING WORKFLOWS

In a conventional toolchain, run-time provenance information detailing the events that occur in the course of executing an application can be gathered by recording logging statements, profiling the program using a tool such as `gprof` [50] or `oprofile` [70], or tracing the application using `gdb` [107] or `strace` [44]. All of this run-time information allows users to analyze their applications for bugs or performance bottlenecks, and thus to make corrections and improvements to the programs. Due to the complex nature of executing a workflow in a distributed environment, it is imperative that the toolchain provide analogous tools that facilitate the profiling and examining of a workflow's execution.

As discussed in Chapter 3, the compiler toolchain in this dissertation depends on Makeflow for managing and executing the actual workflow. Like most other workflow managers, Makeflow generates provenance data detailing the progress of the workflow while executing the tasks in the DAG. Usually this data is recorded in a `makeflowlog` transaction log file, while additional batch system specific information is recording to a `<batch_system>log` file.

This chapter examines a set of profiling utilties that provide different methods of analyzing, monitoring, and reporting a workflow as it is running or after it has been executed. These utilities enable users to analyze the provenance information generated during workflow execution in a consistent and reliable manner in order

to debug their workflows should problems arise and to measure the performance of their applications.

6.1 Workflow Analyzer

The first profiling tool is `makeflow_analyze`, which is a program to parse the `makeflowlog` generated by Makeflow and convert it into a more user-friendly format for analysis. Normally, the provenance log generated by Makeflow is series of event records detailing when a node in the DAG has changed states (e.g. a task goes from the waiting state to the running state). This event record looks something like the following:

```
1336672272295027 17 1 17235 17 1 0 0 0 18
```

In the event record, the first number is the timestamp of the event, followed by the identifier of the node being modified, the new state of the node, and the job identifier associated with the node. After this, the number of nodes in the waiting, running, complete, failed, and aborted states are record, followed the total number of nodes in the DAG. Although this event record is easily parsed by a computer application, it is not very human-readable. For instance, it is not immediately clear from the long how long the workflow took to execute or how many times an individual task failed. To remedy this lack of readability and to convert the log data into more conventional formats, the toolchain includes the `makeflow_analyze` application.

`makeflow_analyze` reads in a `makeflowlog` parses the provenance information, performs some basic statistical computation and data aggregation, and exports this information in a variety for formats. Internally, the profiler uses the `cctools.makeflow.log` recursive descent parser provided by the `python-cctools`

library. This module reads a stream of transaction log events, performs statistical and data processing on the provenance information, and returns a Python data structure for simplified access to the computed workflow profile.

Currently, `makeflow_analyze` can generate plain text, CSV, and JSON output. Figure 6.1 displays a sample output of the `makeflow_analyze` tool in all three formats. The plain text format is normally used for a quick display of the workflow or in conjunction with traditional UNIX text processing tools such as `grep` and `awk`. The comma-separated-values format is useful for importing the provenance data into a spreadsheet such as Microsoft's Excel for manipulation. The JSON format is supported to allow various web and mobile applications to manipulate the provenance information.

Compared to the raw `makeflowlog` event data, the `makeflow_analyze` profiling tool provides a much user-friendlier set of information by performing some data analysis and aggregation while parsing the transaction log. The exported data includes the following information:

- **Start/Stop Times:** It is not clear from the transaction event log when the workflow has started or stopped. This is because the transaction log is node based, and does not record the status of the workflow itself. To address this shortcoming, I modified Makeflow to generate additional provenance information. This augmentation is discussed in detail in Section 7.3. `makeflow_analyze` understands these additional annotations and can report reliable start and stop times.

- **Elapsed Time:** Related to the previous information, the tool provides the total elapsed or running time of the workflow. Again, this is not immediately apparent from the raw transaction log.

```
# Plain Text
log.path = Makeflow.makeflowlog
log.starts = 1336672272.29
log.failures =
log.abortions =
log.completions = 1336672272.6
log.elapsed_time = 0.303680896759
log.percent_completed = 100.0
log.average_tasks_per_second = 59.2727438311
log.current_tasks_per_second = 0
log.estimated_time_left = None
log.state = completed
log.finished = True
log.goodput = 0.57969045639
log.badput = 0
log.nodes.waiting = 0
log.nodes.running = 0
log.nodes.completed = 18
log.nodes.failed = 0
log.nodes.aborted = 0
log.nodes.retried = 0
log.nodes.total = 18

# CSV
Makeflow.makeflowlog,1336672272.29,,,1336672272.6,0.303680896759,100.0,59.2727438311,0,\
    None,completed,True,0.57969045639,0,0,0,18,0,0,0,18

# JSON
{  "log": {
    "estimated_time_left": null,
    "current_tasks_per_second": 0,
    "badput": 0,
    "completions": [ 1336672272.598511 ],
    "starts": [ 1336672272.29483 ],
    "finished": true,
    "elapsed_time": 0.3036808967590332,
    "average_tasks_per_second": 59.272743831110205,
    "state": "completed",
    "percent_completed": 100.0,
    "abortions": [],
    "failures": [],
    "path": "Makeflow.makeflowlog",
    "nodes": {
      "aborted": 0,
      "completed": 18,
      "running": 0,
      "failed": 0,
      "waiting": 0,
      "retried": 0,
      "total": 18
    },
    "goodput": 0.5796904563903809
  }
}
```

Figure 6.1. Sample Workflow Provenance Information by `makeflow_analyze`

This demonstrates the provenance information exported by `makeflow_analyze` *in plain text, CSV, and JSON format.*

- **Task Rates:** The profiling tool also calculates the average tasks that complete per second. If the workflow is still running, a current task rate is calculated and used to compute the estimated time left for finishing the workflow.

- **Goodput/Badput:** `makeflow_analyze` aggregates the amount of computation that yield successful results (goodput) and the computation that ended in failure or abortions (bad put).

- **Node Summaries:** The profiler summaries the number of nodes in the waiting, running, completed, failed, and aborted states.

In addition to reporting information on the whole workflow, `makeflow_analyze` can also be used to profile individual nodes by specifying the `-v` command line option to the application. Users may also filter which nodes to report by specifying the `-F <condition>` command line option. For example, to get all nodes that have failed, a user can pass the argument `-F "node.failures > 0"` and only the nodes that have a failure count greater than 0 will be reported to the user.

A sample of this node provenance information in plain text format is shown in Figure 6.2. For the most part, the node contains much of the same information as the workflow such as elapsed time, goodput, and badput. In addition to this data, the node also includes important information such as the command executed, the input and output files, and its associated job ids, which is data not normally present in Makeflow's transaction log. To facilitate the mapping between tasks and nodes, I augmented Makeflow to embed a copy of the DAG in the transaction log. This is further explained in Section 7.3. All of this information is important because makes it easier for the user to identify which task maps to which node when debugging or profiling a workflow.

```
node.id = 17
node.command = ./stat function.py > function.py.stat
node.original_command = ./stat ./function.py > ./function.py.stat
node.parents =
node.sources = ./stat, ./function.py
node.targets = ./function.py.stat
node.states = 1, 2
node.state = 2
node.timestamps = 1336672272.3, 1336672272.32
node.elapsed_time = 0.0202300548553
node.job_ids = 17235
node.attempts = 1
node.failures = 0
node.abortions = 0
node.goodputs = 0.0202300548553
node.goodput = 0.0202300548553
node.badputs =
node.badput = 0
```

Figure 6.2. Sample Node Provenance Information by `makeflow_analyze`

This is a sample of the type of node specific provenance information provided by `makeflow_analyze` *in plain text format.*

Figure 6.3 is an example of a creative use of the provenance information exported by the `makeflow_analyze` utility. For the CSE 60333 Mobile Application Development course, Michael Albrect, Patrick Donnelly, and I built a small Android application called *LuLuLua* that was basically a framework for building small mobile applets using the Lua [58] programming language. One of these applets was a Makeflow monitor application that periodically pulled a JSON file generated by the `makeflow_analyze` profiler and displayed a visual summary of the workflow's progress based on the exported provenance information.

Because the Makeflow transaction journal is so terse, the `makeflow_analyze` profiling utility is an important and useful tool for extracting information about a workflow. Since the profiler tool processes and aggregates the provenance information from the raw Makeflow transaction log, the `makeflow_analyze` tool saves

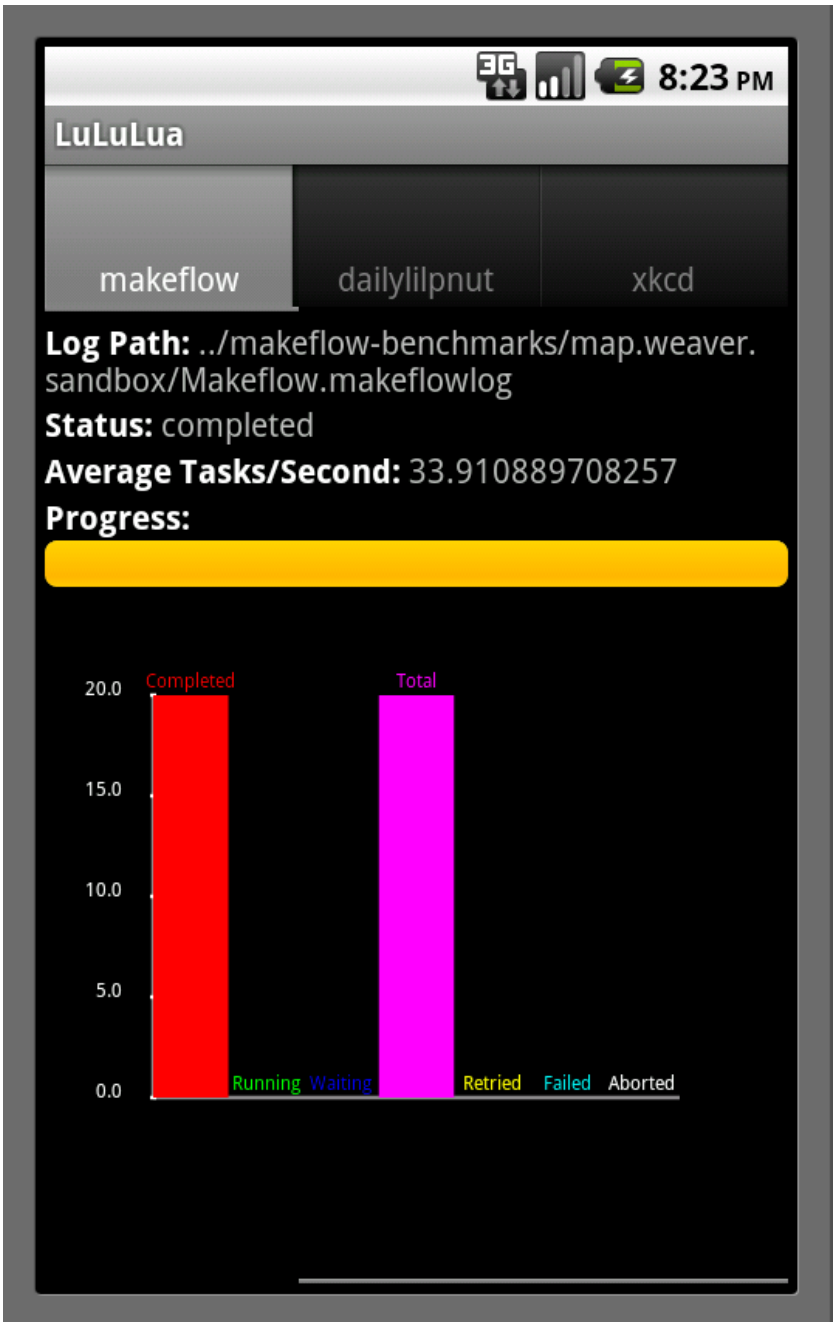


Figure 6.3. Workflow Progress with LuLuLua Android Application.

In this screenshot, the LuLuLua Android Application is retrieving a JSON file exported by the `makeflow_analyze` *utility and displaying the progress of the Makeflow along with other provenance information on a mobile device.*

users from having to create ad hoc programs for computing this information. By providing multiple export formats, the profiler allows for traditional command-line processing, spreadsheet manipulation, and even web and mobile visualization.

6.2 Workflow Monitor

The next profiling tool is the `makeflow_monitor`. The current implementation of Makeflow does not provide the user a display of its progress and thus it is difficult for a user to know how far along the workflow is in terms of execution. Fortunately, Makeflow records event information to a transaction log as discussed above, which can be processed to get the current status of the workflow.

As with the `makeflow_analyze` utility, the `makeflow_monitor` application uses the `cctools.makeflow.log` module from the `python-cctools` library to parse and process `makeflowlog` generated by Makeflow. Once the transaction log is processed and the statistical information is computed, the `makeflow_monitor` reads the returned Python data structure and outputs the status of the workflow to the active console or terminal. Normally, the transaction log to monitor is passed via command-line arguments to `makeflow_monitor`, which can accept multiple `makeflowlogs`. For convenience, the utility also has the ability to watch a directory for new transaction logs and will automatically parse and display any logs that it detects.

Figure 6.4 shows an example of the console output of the `makeflow_monitor` utility. In this example, the progress of two workflows are displayed. The first workflow appears at the top of the output and has a "Completed" status. The `makeflow_monitor` utility reports it's start time, elapsed execution time, the average tasks completed per minute, and a summary of all the task states. The second

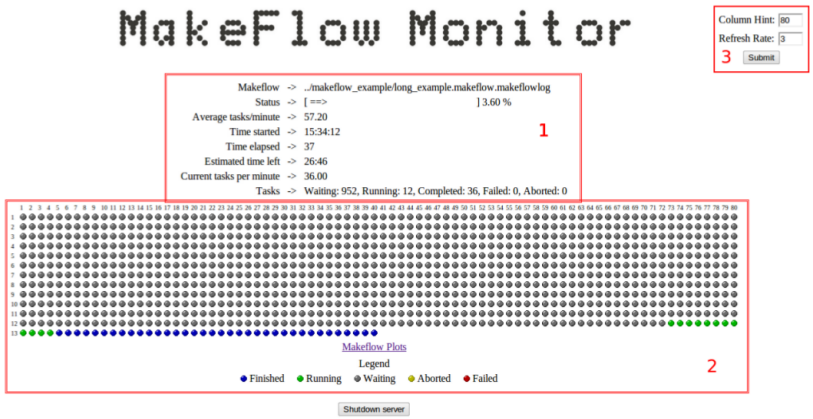
```
            Makeflow: nostash.1.makeflowlog
              Status: Completed
        Time Started: 09:39:46
        Time Elapsed: 32:12
Average Tasks/Minute: 3104.44
               Tasks: Waiting: 0, Running: 0, Completed: 100000, Failed: 0,
                      Aborted: 0, Retried: 0, Total: 100000

            Makeflow: nostash.2.makeflowlog
              Status: [===================================>            ] 75.02%
        Time Started: 10:12:08
        Time Elapsed: 25:42
Average Tasks/Minute: 2917.43
Current Tasks/Minute: 2853.65
 Estimated Time Left: 08:45
               Tasks: Waiting: 24968, Running: 15, Completed: 75017, Failed: 0,
                      Aborted: 0, Retried: 0, Total: 100000
```

Figure 6.4. `makeflow_monitor` Console Output

In console mode, the `makeflow_monitor` *periodically outputs a variety of progress information such as the workflow status, the elapsed time, etc. directly to the user's terminal. It has options such as sorting by workflow progress, hiding completed workflows, and watching set of directories for new Makeflow logs.*

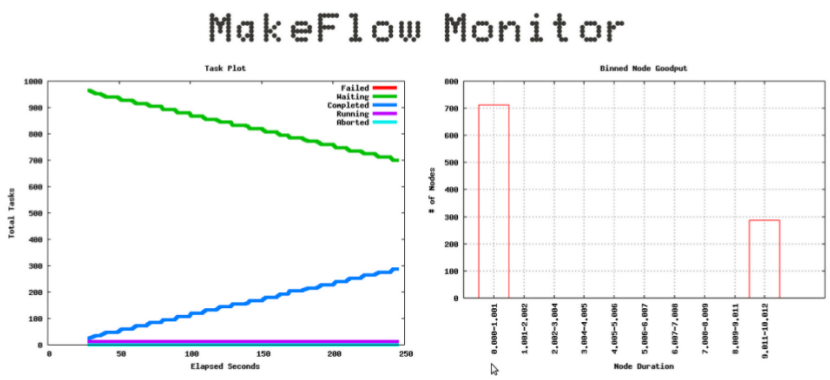
workflow is still executing, so the status shows a progress bar and percentage. In addition to the fields mentioned for the first workflow, `makeflow_monitor` also reports the current tasks completion rate and the estimated time left. As mentioned previously the `cctools.makeflow.log` performs all of these computations, so the information displayed in the `makeflow_monitor` is the same information exported by the `makeflow_analyze` utility which uses the same library.

While the console output provided by `makeflow_monitor` is useful and sufficient for quickly assessing the progress of a workflow, some users perform a more graphical and detailed user interface. To support a more graphical monitoring of workflows, I hired and advised Samuel Lopes, an undergraduate Computer Science student, for a semester to create a graphical and more user-friendly version



(a) Workflow Overview

(b) Node Information



(c) Workflow Plots

Figure 6.5. `makeflow_monitor` Web Output

The web version of the `makeflow_monitor` *displays a dashboard showing the overall progress of the workflow. By selecting a node from the dashboard, the individual task information is displayed. The web site also provides a few graphs illustrating the different aspects of the workflow's execution.*

of the `makeflow_monitor`. The result of this undergraduate research experience is the web version of the monitoring utility as shown Figure 6.5.

Samuel took the original `makeflow_monitor` and used modules from the Python standard library to embed a small web server. In his new version, when the user starts `makeflow_monitor` with the `-t webserver` option a HTTP server is started on the local machine. For security reasons, the web site served by the HTTP server requires basic HTTP authentication. When the server is started, the username and password is automatically generated and printed to the console for the user to copy and paste into the web browser.

Once users access the web site served by the modified `makeflow_monitor`, they are greeted with an overview of the workflow as shown in Figure 6.5a. The textual information displayed here is nearly identical to the console output of the normal `makeflow_monitor`. Beneath the overview is a dashboard of colored circles representing the tasks in the workflow. Blue means the task has completed, while green means the task is running, and gray indicates that task is in the waiting state. Clicking on one of the task circles takes the user to a page with the node information associated with the task as shown in 6.5b. This information is similar to the node specific data exported by `makeflow_analyze`. Finally, the website also includes a couple of graphs based on the provenance data as demonstrated in 6.5c. The graph on the left shows how the various task states change over the execution of the workflow, while the second graph is a histogram of the task execution times.

Because Makeflow does not provide a user-friendly way of monitoring the progress of a currently workflow, the toolchain provides `makeflow_monitor` as a means of assessing the status of the workflow. By default, this profiling util-

ity can monitor multiple transactions logs or even a directory and displays the progress of the workflows to the console. For graphical monitoring of a workflow, I worked with Samuel Lopes to produce a version of the profiler that presented the progress information through a web site. Currently, Samuel's web extension of the `makeflow_monitor` is available separately from the original utility. There are plans, however, to integrate the two in the near future.

6.3 Workflow Reporter

The final profiling utility is `makeflow_report`. Once again, this application uses the `cctools.makeflow.log` modules from the `python-cctools` package to read and parse Makeflow transaction logs. After the raw event data is processed, this utility performs some high-level statistical analysis such as determining the fastest and slowest tasks, the mean task execution time, and the median task execution time. It also sorts the tasks and creates a histogram of the execution times to allow the user to examine the running-time characteristics of the workflow. All of this profiling information is gathered, formatted, and then outputted to the console as plain text.

In addition to providing task-based profiling information, the `makeflow_report` utility can also display profiling statistics for Weaver `Abstractions` and `Functions` if the transaction log contains symbolic annotations. Normally, the Makeflow DAG only contains the rules that specify the workflow. By using the `-g` flag, the user can instruct the Weaver compiler to embed symbolic annotations into the DAG, similar to how GCC can include debugging symbols in object code. More details on how symbolic annotations are implemented and used throughout the toolchain are discussed in Section 7.3.

With symbolic annotations turned on, the profiler will aggregate provenance data for specific `Abstractions` or `Functions` and display the average, median, slowest, and fastest execution times for the tasks associated with these symbols along with the tasks states of the nodes associated with the symbol.

An example report generated by `makeflow_report` is presented in Figures 6.6, 6.7, 6.8, 6.9 (this is a single report split into separate parts for formatting purposes and increased readability).

```
Makeflow Log Summary: /tmp/pbui-weaver-tests/map/Makeflow.makeflowlog
======================================================================

               Status: Completed
         Time Started: 17:00:18
         Time Elapsed: 33
 Average Tasks/Minute: 32.21
 Current Tasks/Minute: 0.00
  Estimated Time Left: None
     Good Computation: 287.93
      Bad Computation: 0.00


 Tasks Summary
 -------------

     Total Tasks: 18
   Tasks Waiting: 0 (0.00%)
   Tasks Running: 0 (0.00%)
    Tasks Failed: 0 (0.00%)
   Tasks Aborted: 0 (0.00%)
 Tasks Completed: 18 (100.00%)
```

Figure 6.6. `makeflow_report` Sample (Overview)

The top part of the generated report, as shown in Figure 6.6 displays the summary of the workflow. Unsurprisingly, this information is similar to that found in `makeflow_analyze` and `makeflow_monitor`. Instead of a long line for the

task states as found in the two tools, the `makeflow_report` utility breaks down the task states in tabular format.

The next portion of the report is displayed in Figure 6.7. Here, a statistical profiling of all the tasks in the workflow is presented. Specifically, the average, median, slowest, and fastest task execution times are reported. Additionally, the details of the three median, slowest, and fastest nodes are also provided. By displaying the node command and symbol (if available) it is possible for the user to not only determine what the fastest task execution time is, but what tasks achieved that time. This allows the user to discover any possible performance bottlenecks and make adjustments to the workflow.

The task profiling is followed by a histogram of the task execution times as presented in Figure 6.8. The purpose of this chart is to allow the user to see the spread of their tasks in terms of execution times. This graphical display makes it easy for users to quickly identifier if there are any outliers such as stragglers taking an unusable amount of time.

The last part of the report is shown in Figure 6.9 and is only generated if symbolic annotations are present in the `makeflowlog`. In this portion of the report a summary of all of the task states for each included symbol (i.e. Weaver `Abstraction` or `Function`) is displayed. This is followed by a reporting of the task statistics for each symbol. The benefit of having symbolic annotations and this provenance reporting is that the user can determine which `Abstractions` are problematic. That is, users can analyze their workflow in terms of high-level components rather than merely looking at low-level tasks. This is advantageous because the original Weaver specification is defined in terms of these high-level symbols and not individual tasks.

```
Tasks Profiling
---------------

Average Task Time: 15 +/- 02

 Median Task Time: 15

   1. NODE    9
      TIME    14
      SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
      COMMAND ./stat.sfx /home/pbui/src/research/weaver/weaver/__init__.py >
              __init__.py.stat
   2. NODE    8
      TIME    15
      SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
      COMMAND ./stat.sfx /home/pbui/src/research/weaver/weaver/options.py >
              options.py.stat
   3. NODE    7
      TIME    17
      SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
      COMMAND ./stat.sfx /home/pbui/src/research/weaver/weaver/compat.py >
              compat.py.stat

Slowest Task Time: 33

   1. NODE    1
      TIME    30
      SYMBOL  Map[0](/usr/bin/convert {IN} {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000001)
      COMMAND ./convert.sfx /usr/share/pixmaps/xterm_32x32.xpm xterm_32x32.jpg
   2. NODE    2
      TIME    28
      SYMBOL  Map[0](/usr/bin/convert {IN} {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000001)
      COMMAND ./convert.sfx /usr/share/pixmaps/parcellite.xpm parcellite.jpg
   3. NODE    3
      TIME    25
      SYMBOL  Map[0](/usr/bin/convert {IN} {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000001)
      COMMAND ./convert.sfx /usr/share/pixmaps/xterm-color_48x48.xpm xterm-
              color_48x48.jpg

Fastest Task Time: 02

   1. NODE    17
      TIME    02
      SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
      COMMAND ./stat.sfx /home/pbui/src/research/weaver/weaver/function.py >
              function.py.stat
   2. NODE    16
      TIME    04
      SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
      COMMAND ./stat.sfx /home/pbui/src/research/weaver/weaver/data.py >
              data.py.stat
   3. NODE    15
      TIME    05
      SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
      COMMAND ./stat.sfx /home/pbui/src/research/weaver/weaver/nest.py >
              nest.py.stat
```

Figure 6.7. `makeflow_report` Sample (Tasks Profiling)

```
Tasks Histogram
---------------

     02 [xxxx                                ] 11.11%
     05 [xxxx                                ] 11.11%
     08 [xxxx                                ] 11.11%
     11 [xxxx                                ] 11.11%
     14 [xxxx                                ] 11.11%
     17 [xxxxxx                              ] 16.67%
     20 [xx                                  ]  5.56%
     23 [xx                                  ]  5.56%
     26 [xx                                  ]  5.56%
     29 [xx                                  ]  5.56%
     32 [xx                                  ]  5.56%
     35 [                                    ]  0.00%
```

Figure 6.8. `makeflow_report` Sample (Tasks Histogram)

Because of the myriad number of problems and issues that may arise when executing a workflow on a distributed system, it is important to have the ability profile and analyze the workflow application. The toolchain addresses this difficulty by including tools to process the provenance information, monitor the status of the workflow, and generate a report summarizing the key aspects of the workflow.

The impact of these tools are increased productivity and improved workflow analysis. For instance, the distilled provenance information exported by `makeflow_analyze` utility was used to automate the generation of graphs and plots for not only this dissertation but various publications that utilize the toolchain. Likewise, while executing the many benchmarks and tests for this dissertation, I regularly employed the `makeflow_monitor` utility to track of the status of the workflows and to estimate how much time left was required to complete the tests. Moreover, whenever I ran into problems or needed to compare two workflows, I used the `makeflow_report` tool to get a break-down of the workflows' tasks in order to identify bottlenecks and to get a sense of the overall pattern of a workflow.

```
Symbols Summary
---------------

Total Symbols: 2

   1. SYMBOL  Map[0](/usr/bin/convert {IN} {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000001)
        TIME  02:27
       TASKS  5
              Waiting: 0 (0.00%)
              Running: 0 (0.00%)
               Failed: 0 (0.00%)
              Aborted: 0 (0.00%)
            Completed: 5 (100.00%)
   2. SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
        TIME  02:27
       TASKS  13
              Waiting: 0 (0.00%)
              Running: 0 (0.00%)
               Failed: 0 (0.00%)
              Aborted: 0 (0.00%)
            Completed: 13 (100.00%)

Symbols Profiling
-----------------

   1. SYMBOL  Map[0](/usr/bin/convert {IN} {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000001)
         Average Time: 28 +/- 01
          Median Time: 28
         Slowest Time: 33
         Fastest Time: 22

   2. SYMBOL  Map[1](/usr/bin/stat {IN} > {OUT},/tmp/pbui-weaver-tests/map/_Stash/0/0/0/0000003)
         Average Time: 11 +/- 01
          Median Time: 11
         Slowest Time: 19
         Fastest Time: 02
```

Figure 6.9. `makeflow_report` Sample (Symbols Profiling)

These profilers were instrumental in analyzing and understanding the nature of sophisticated workflows such as the transcoding hierarchical benchmark in Chapter 4. While using `makeflow_monitor` to track the transcoding benchmarks, I discovered that different file types exhibited very different running times. Using the `makeflow_report` utility I quickly identified the video transcoding as the chief bottleneck. This in turn lead me to test both small and big workflows and to plot the current task completion rate of each file type in order to further understand the impact of interleaving different types of tasks. Without the profiling tools,

it would have been difficult for me to recognize the behavior of the transcoding workflows and to adjust the metrics I was measuring and investigating.

Altogether, these profiling tools provided by the toolchain increase productivity by enaling users to effectively analyze workflows, monitor their progress, and identify any problems that occurred during execution.

CHAPTER 7

MANAGING WORKFLOWS

As discussed in previous chapters, the compiler toolchain in this dissertation relies on Makeflow [3] as its run-time system. Because it is a portable workflow manager that supports Makefile [43] syntax, Makeflow serves as a convenient and straightforward tool for constructing large distributed data intensive scientific workflows that can be executed on a variety of distributed systems such as Condor [113], SGE, and WorkQueue [22]. It has been used to build a variety of research and production level scientific workflows with great success [3, 21, 23, 66, 114], and thus is a capable run-time target for the compiler toolchain.

This chapter examines a few modifications that extend Makeflow's current capabilities and enable new additional features that enhance the usability and in some cases the performance of the toolchain and the workflows it generates. The following augmentations to Makeflow are discussed:

1. **Local Variables:** The current implementation of Makeflow only supports global environmental variables. There are situations where it would be desirable to task-specific variables to define items such as batch system specific parameters or task specific configuration settings. Section 7.1 presents how the Makeflow parser was redesigned to support this use case.

2. **Nested Makeflows:** Currently, Makeflow has no knowledge of whether or not a task is actually a recursive or nested Makeflow. Because of this, it cannot make decisions such as setting resource constraints for child workflows. Section 7.2 details how I introduced the `MAKEFLOW` operator to denote that a task was indeed another workflow, and then provides an example how this information can be used to control nested resources.

3. **Provenance and Annotations:** As discussed in Chapter 6, Makeflow records events to a transaction log during execution. Unfortunately, this journal is limited and not sufficient for the higher-level profiling capabilities desired by the toolchain. Section 7.3 presents the additional provenance information emitted and supported by Makeflow that enables for sophisticated debugging and track of Makeflow DAGs.

4. **Garbage Collection:** Finally, Section 7.4 revisits the problem of intermediate files by exploring the use of various garbage collection methods during the execution of a workflow managed by Makeflow. I perform a benchmark of these methods on three different filesystems to examine the performance impact of garbage collection on workflows with many intermediate files.

7.1 Local Variables

In the current implementation of Makeflow, all variables defined in Makeflow are simply environmental variables. When the Makeflow parser comes across a variable it simply performs a `setenv` to store the variable in the environment of the current process. This is convenient as it allows for batch jobs to access these variables by simply inheriting the environment of workflow manager and absolves Makeflow from having to track the variables itself.

Unfortunately, the current parser design means that it is difficult to set task-specific variables. For instance, one command may require a certain variable to be set to one value and another command may need another setting. Because the variables in Makeflow are global it is not possible to set a task-specific version for each rule in the current implementation.

A possible work around is to simply use the `env` command to set environment variables as a part of executing a task's shell command. Consider the following set of Makeflow rules:

```
out.0: in.0
    env VAR=0 command in.0 out.0

out.1: in.1
    env VAR=1 command in.1 out.1
```

In both rules, the `VAR` variable will be set to local values and made available to the batch job process. That is when the first command executed it can access `VAR` and it will have the value 0, while the second command would retrieve 1 from the variable `VAR`. Unfortunately, this is insufficient if the variable is needs to be made available as part of the shell comand or to Makeflow itself. For instance, Makeflow supports the `BATCH_OPTIONS` variable that allows the user to send any batch system specific parameters such as Condor requirements or SGE command-line arguments. Because the `env` is executed as part of the batch job, the variable definitions are not available to the caller, Makeflow, and thus this mechanism would not support setting the `BATCH_OPTIONS` parameter.

One possible modification to support task-specific variables is to simply have the variables be bound when tasks are parsed by Makeflow. For instance, consider the following Makeflow DAG:

```
VAR=0
out.0: in.0
    command $VAR in.0 out.0

VAR=1
out.1: in.1
    command $VAR in.1 out.1
```

In this proposal, when the first rule is parsed it would store `VAR` as 0 which was set on the line before it. The second rule would store `VAR` as 1. This would satisfy the need to set task-specific variable definitions for both the process and Makeflow. The problem with this method, however, is that it removes Makeflow's declarative syntax. Instead of being context-free, the meaning of variable depends on the *order* in which variables and tasks are defined, which is an undesirable characteristic and breaks compatibility with the original Make.

An alternative approach to this is to introduce a minimal form lexical scoping [80, 109] such that each task has its own variable environment separate from the global variable environment. In this approach there are now three levels of variable scoping:

1. **Local Task**: Variables defined at this scope are only available to the task being defined. Consider the following Makeflow rule:

   ```
   out: in
       @LOCAL_VAR = cat
       $LOCAL_VAR in > out
   ```

   In this example, a task-specific variable is defined by prefixing the variable assignment statement with `@` and placing it underneath the file specification component of the task rule. This is done to maintain syntax compatibility

with Make, but not semantic compatibility (i.e. the rule would parse fine in Make, but would not necessarily yield the same output).

In GNU Make, this situation is handled by supporting target-specific variable assignments. The above rule look like this in GNU Make:

```
out: LOCAL_VAR = cat
out: in
    $LOCAL_VAR in > out
```

In GNU Make, we specify the output target and set the variable we wish to define. Now any rule used to generate this output target will have this local variable. Unfortunately, supporting this target-specific variable assignment syntax would have required invasive modifications to Makeflow (i.e. would need to switch to a two-pass parser instead of a single-pass as it is now). Moreover, semantically, Makeflow is task-centric rather than target-centric, which means it makes more sense for the user to associated a variable setting with a whole task than individual targets. Thus the above `@` syntax for specifying task local variables was introduced.

It should be noted that this target-specific variable syntax is only supported by GNU Make not any of the other Make systems such as BSD Make.

2. **Global DAG**: Variables defined at this scope are available to the whole DAG, including the task nodes:

```
GLOBAL_VAR=cat
out: in
    $GLOBAL_VAR in > out
```

3. **Process Environment**: These are read-only variables that can be fetched from the process environment (i.e. using `getenv`).

In the previous version of Makeflow, all variables were set and fetched from the global process environment. With this new implementation, the original process environment is basically left alone. When a global variable is defined, it is stored in a DAG environmental table which all tasks share, while local variables are stored in a table associated with that particular node. To resolve a variable, the local node table is searched, then the global DAG, and then the process environment. This means that the task variables can shadow global variables (i.e. have the same name, but different values) but always take precedence. That is if a variable is defined in both the global and task environment, then the value from the task environment will be used rather than the global one.

Because variables are no longer stored in Makeflow's process environment (and thus implicitly exported), the variable definitions are only available to Makeflow itself and in parsing the rules. To allow for executing batch jobs access to the variables the `export` operator was introduced as a mechanism for enabling the user to explicitly state which variables to export so that the values are available to a running batch job process. The following code demonstrates the use of the `export` command:

```
GLOBAL_VAR=1
out:
    echo $$GLOBAL_VAR > out
export GLOBAL_VAR
```

In this example, the `GLOBAL_VAR` variable is exported such that the task shell command can access it. Internally, this is implemented by maintaining a global DAG-wide export list. Before any batch job is executed, all of the variables in this

export list is set in the environment. This implementation of the `export` operator matches the syntax and semantics of the `export` command found in GNU Make.

To support the definition and exporting of variables, the compiler was updated to add the `Define` and `Export` functions. The former allows users to define a global variable, while the latter is used to tell Makeflow which variables to export. For task specific variables, the user may pass in an dictionary containing variable definitions to a `Function` or `Abstraction` using the `environment` keyword argument.

```
# Weaver Script
Define('MYVAR1', 1)
Export(['MYVAR1', 'MYVAR2'])

env = ParseFunction('env > {OUT}', enviroment={'MYVAR2': 2})
env(outputs='env0.txt')
env(outputs='env1.txt', environment={'MYVAR3': 3})

# Makeflow DAG
env0.txt: /usr/bin/env
        @MYVAR2=2
        /usr/bin/env > env0.txt
env1.txt: /usr/bin/env
        @MYVAR3=3
        @MYVAR2=2
        /usr/bin/env > env1.txt
MYVAR1=1
export MYVAR1
export MYVAR2
```

Figure 7.1. Weaver Variable Example

The top portion of this example demonstrates how to define and export variables in Weaver, while the bottom portion shows the Makeflow DAG generated by the compiler when processing the Weaver script.

Figure 7.1 provides an example of how `Define` and `Export` are used along with the `environment` keyword argument in Weaver to specify variables in the Makeflow DAG. At the top is the Weaver Script that defines `MYVAR1` and exports `MYVAR1` and `MYVAR2`. Next it defines the `env Function` that includes an `environment` dictionary that defines the `MYVAR2` variable. After this, we call the `env` variable twice. The first with just an `outputs` argument and the second time with an additional environment table that defines the `MYVAR3` variable. Below the Weaver script is the Makeflow DAG generated by the DAG. As can be seen the DAG rules match the Weaver specification: `MYVAR1` is defined as a global variable, the `MYVAR1` and `MYVAR2` variables are exported, and each task has the appropriate set of local variables associated with it.

As mentioned earlier, one of the motivations for having support for task-specific variables is to allow users to specify batch-specific options for particular tasks. For instance, one task may require one set of machines while another may require machines with a specific amount of memory. In the current version of Makeflow, such resource specifications can be only be set globally through the `BATCH_OPTIONS` environment variable. With the new support for local task variables, users can now specify batch options for individual tasks as show in Figure 7.2. In this example, we create three tasks that require three different Condor machine groups. To do this, we simply create the batch options string and then pass it to the function via an environmental dictionary as shown previously. When the generated Makeflow is executed using the Condor batch system, then these requirements will be utilized to schedule the tasks to the desired set of machines.

Overall, the implementation of local variables satisfies our goal of enabling users to set task specific variables while minimizing the amount of changes to

```
# Weaver Script
uname = ParseFunction('uname -a > {OUT}')

for group in ['disc', 'ccl', 'gh']:
    options = 'requirements = MachineGroup == "{0}"'.format(group)
    env     = {'BATCH_OPTIONS': options}
    uname(outputs='uname.{0}'.format(group), environment=env)

# Makeflow DAG
uname.disc: /bin/uname
        @BATCH_OPTIONS=requirements = MachineGroup == "disc"
        /bin/uname -a > uname.disc
uname.ccl: /bin/uname
        @BATCH_OPTIONS=requirements = MachineGroup == "ccl"
        /bin/uname -a > uname.ccl
uname.gh: /bin/uname
        @BATCH_OPTIONS=requirements = MachineGroup == "gh"
        /bin/uname -a > uname.gh
```

Figure 7.2. Weaver Batch Options Example

This example shows how to set the `BATCH_OPTIONS` *environment variable for individual tasks. The top part is the Weaver source, while the bottom is the Makeflow DAG generated by the compiler. As can be seen, the local task variable mechanism allows for each task to require a separate* `MachineGroup`*.*

Makeflow itself. As noted previously, this modification is syntactically compatible with GNU Make but not semantically. This is unfortunate, but using Make's target-specific variable assignment mechanism would require major changes the Makeflow's current implementation. Moreover, the target-specific syntax did not match Makeflow's task-centric model. That said, the internal mechanism in how variables are defined and resolved are basically equivalent and thus it would be conceivable that the two syntaxes could be reconciled in a future work. The main point is to allow users to set local task variables such as `BATCH_OPTIONS` and to show that the toolchain can take advantage of this feature, which the examples above demonstrate.

7.2 Nested Makeflows

The second modification to Makeflow is to turn nested Makeflows into first-class objects in the DAG language. This means that after parsing the DAG, Makeflow should be aware if a particular task is a nested invocation of Makeflow. Knowing this information will allow Makeflow to make run-time adjustments to things such as resource allocation. Moreover, having Makeflow as a first class object also makes it easier for the toolchain utilities to manipulate and modify a hierarchy of Makeflows.

Normally, if a user wishes to nest a Makeflow, they simply construct a task whose command is `makeflow <dag_path>`. This, however, does not allow the workflow manager to make intelligent decisions regarding resource allocation since it cannot distinguish between a rule for a nested Makeflow and a rule for a normal task. Conceivably, Makeflow could search the shell command to detect if makeflow is called, but this would not be too error-prone and fragile, especially if command line arguments are present in the shell command.

To address this problem and make Makeflow aware of child workflows, I introduced the `MAKEFLOW` command operator. When a user wishes to have a task execute a nested Makeflow, the user specifies the following as the command:

```
MAKEFLOW "<dag_path>" "<dag_work_dir>" "<dag_wrapper>"
```

The first argument after the `MAKEFLOW` keyword is the path to the DAG file. The next argument is the path to the workspace to use. The final argument is used for any wrapper command required such as Parrot [112]. For instance, if the nested Makeflow's DAG is called "Makeflow.nested", is to be executed in the current directory, and requires Parrot, then the command would look like this:

```
MAKEFLOW "Makeflow.nested" "." "parrot_run"
```

For command-line arguments, I introduced new environmental variables such as `MAKEFLOW_BATCH_QUEUE_TYPE` and `WORK_QUEUE_MASTER_MODE` to allow users to set specific Makeflow options for the nested workflows. This, of course, utilizes the local task variable mechanism discussed in the previous section.

Implementing the `MAKEFLOW` operator in the current Makeflow program required minimal modifications to the parser. In the augmented version of Makeflow, when the above `MAKEFLOW` syntax is encountered, it is translated to the following task shell command:

```
cd <dag_work_path> && <dag_wrapper> makeflow <dag_path>
```

Additionally, the node is marked as a Makeflow job to allow the workflow manager to make resource allocation decisions during execution.

As a proof-of-concept of this resource management ability, I implemented a mechanism in which the maximum number of local batch jobs for each nested Makeflow would be proportional to the maximum number of concurrent sub-workflows in the DAG. To do this, I modified Makeflow's `dag_width` function which was originally implemented by Kevin Partington, an undergraduate, using an algorithm I had designed. The purpose of this function is to determine the approximate maximum width of a directed acyclic graph. If we can determine this, then we would know the maximum amount of concurrency possible in our workflow, since the width represents the largest number of independent tasks.

The algorithm used to compute this width is outlined in Figure 7.3. The idea is to divide the DAG into levels such that in each level every node is independent of each other. That is all the nodes in a particular level can be executed in parallel.

```
# Return the maximum width of the DAG
def dag_width(dag):
    # Count the number of children each node has.
    for node in dag.nodes:
        for file in node.source_files:
            parent = file.parent
            parent.children++

    # Determine leave nodes based on number of children.
    leaves = []
    for node in dag.nodes:
        node.remaining = node.children
        if node.children == 0:
            leaves.append(node)

    # Starting from the bottom (i.e. leaves), traverse graph and
    # mark levels such that a parent node is always one level
    # higher than its children.  Keep track of the maximum level.
    max_level = 0
    while leaves:
        node = leaves.pop()
        for file in node.source_files:
            parent = file.parent
            if parent.level < node.level + 1:
                parent.level = node.level + 1

            max_level = max(max_level, parent.level)

            parent.remaining--
            if parent.remaining == 0:
                leaves.append(parent)

    # Tally up all of the nodes in each level.
    levels = []
    for node in dag.nodes:
        levels[node.level]++

    # The level with the most nodes represents the widest part of
    # the DAG, so return the maximum level.
    return max(levels)
```

Figure 7.3. `dag_width` Algorithm

The basic idea behind this algorithm is to traverse the graph from the bottom up, setting the node's level, and tracking the high level. Once the nodes are marked, we simply count of the number of nodes at each level, and then return the value of the level with the most nodes.

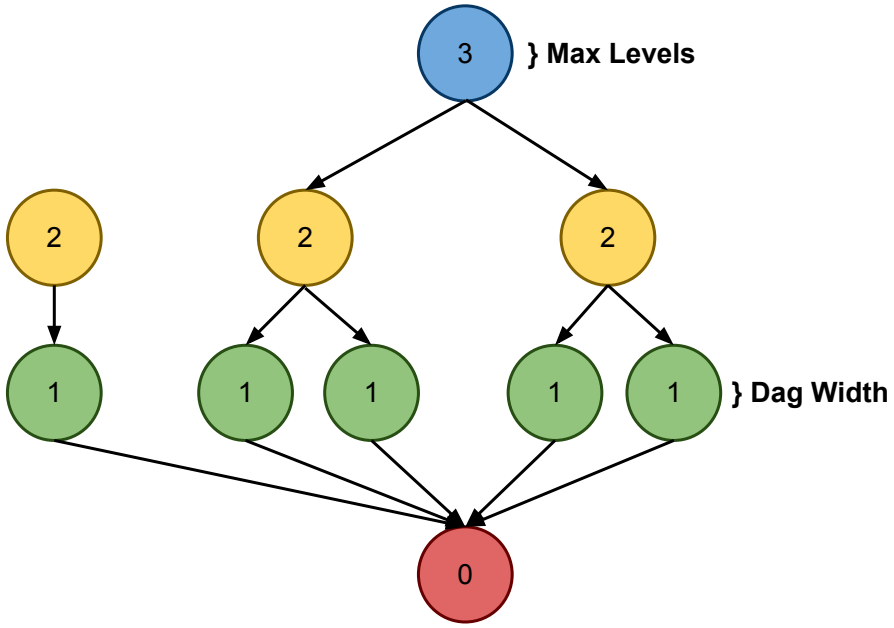


Figure 7.4. `dag_width` Illustration

In this algorithm, we begin marking levels from the bottom and traverse upwards such that the parent's level is always one more than its children. The level with the most nodes is considered the width of the DAG, while the highest level corresponds to the height of the DAG.

This is done by first marking all of the nodes with the number of children and determining the leave nodes. For every leaf node, we update its parent nodes to ensure that the parent's level is one more than the current leaf. Likewise, we keep track of the maximum level and update the list of leaves when the parent node's count of remaining children reaches zero. Once all of the nodes in the DAG are separated into levels, then we simply need to count of all the nodes in each level and then find the level with the most nodes to determine the highest amount of concurrency available in our workflow.

Figure 7.4 illustrates the `dag_width` algorithm. Going from the bottom up, we start by marking the first leaf as level 0 and then update all of its parents with level

1. This continues up the graph. Afterwards, we look at each level and count all of the nodes in a particular level. In this case, level 1 has 5 nodes and is the widest part of the graph. This means that the maximum concurrency for the workflow is 5, since at any one time, we can have at most 5 nodes running in parallel. Knowing the width of the DAG allows us to allocate resources appropriately.

In the context of nested Makeflows, I performed a slight variation to the algorithm described, such that it only computed the maximum number of concurrent nested workflows (rather than all tasks). With this number, I then implemented a mechanism that would allocate an appropriate amount of local cores to the nested workflows. Whenever I detected a nested Makeflow that was to be run locally, I checked the maximum number of concurrent nested DAGs and divided the current max local jobs by that `dag_width`. For example, if the maximum number of local jobs is 16 and I have a maximum of 4 nested concurrent workflows, then each of the child workflows would be allowed $16/4 = 4$ local jobs. This is significant because in the previous version of Makeflow, no resource restraints would be placed on the nested workflows, which means the local machine could be swamped with $16 \times 4 = 64$ local jobs, which could have a severe performance impact if the tasks are compute intensive.

The point here is that having nested Makeflows as first-class objects allows the workflow manager to make intelligent resource allocation decisions for hierarchical workflows. In this case, we limit the number of local batch jobs for nested workflows to prevent overloading the local machine. Having `MAKEFLOW` as a keyword in the DAG also allows for the toolchain utilities to be aware of the DAG's relationship to other workflows. This is important for tools such as `makeflow_link`, which can recursively link nested workflows for the user. Although use of this informa-

tion is limited to this proof-of-concept resource allocation mechanism and to a few of the toolchain utilities, having the workflow manager aware of nested Makeflows is a necessary pre-cursor to enabling future work involving more sophisticated resource management and scheduling.

7.3 Provenance and Annotations

The third set of modifications to Makeflow involve enhancing the provenance information recording in the workflow manager's transaction journal. As explained in Section 6.1, Makeflow records a series of event logs to a journal file during the execution of a workflow. This event record is a series of numbers denoting the time of the event, the identity of the node, the state of the node, the job identifier of the node, and an accounting of all the node states in the graph. The primary purpose of the log is to allow Makeflow to restart or recover from a failed or aborted execution. Whenever Makeflow is started, it will attempt to recover the journal and update its internal DAG to match the state of the journal. This way tasks that have completed will be skipped during this new attempt, allowing users to continue a failed or aborted workflow without having to redo all of the previous work.

While the event log in the current Makeflow is suitable for recovery, it is not sufficient for the tracking, monitoring, and analyzing features the toolchain utilities implement. To enable these enhanced profiling tools, I modified Makeflow in the following ways:

1. **Embed DAG**: The information in the event records stored in the transaction log only refer to nodes by their identifier number. Based on this number alone, it is not clear what command was executed or what the input and

```
# NODE  2       /usr/bin/env > env1.txt
# SYMBOL        2       /usr/bin/env > {OUT}
# PARENTS       2
# SOURCES       2       /usr/bin/env
# TARGETS       2       env1.txt
# COMMAND       2       /usr/bin/env > env1.txt
# NODE  1       /usr/bin/env > env0.txt
# SYMBOL        1       /usr/bin/env > {OUT}
# PARENTS       1
# SOURCES       1       /usr/bin/env
# TARGETS       1       env0.txt
# COMMAND       1       /usr/bin/env > env0.txt
# NODE  0       /usr/bin/stat /etc/hosts > hosts.stat
# SYMBOL        0       /usr/bin/stat {IN} > {OUT}
# PARENTS       0
# SOURCES       0       /usr/bin/stat   /etc/hosts
# TARGETS       0       hosts.stat
# COMMAND       0       /usr/bin/stat /etc/hosts > hosts.stat
# STARTED       1337095010326731
1337095010326974 2 1 5465 2 1 0 0 0 3
1337095010327204 1 1 5466 1 2 0 0 0 3
1337095010340777 1 2 5466 1 1 1 0 0 3
1337095010340950 0 1 5469 0 2 1 0 0 3
1337095010341021 2 2 5465 0 1 2 0 0 3
1337095010370866 0 2 5469 0 0 3 0 0 3
# COMPLETED     1337095010370898
```

Figure 7.5. Modified Makeflow Transaction Journal Example

The modified Makeflow transaction journal now contains **(1)** *embedded DAG,* **(2)** *workflow status records,* **(3)** *symbolic annotations. All of this additional provenence information allows for the profiling utilities to perform richer and more accurate analysis.*

output files were. To figure this out, one would have to parse the original DAG and the transaction journal to match up node *ID* to task properties. To having to parse two files and perform this mapping, I modified Makeflow so that it embeds the DAG inside the transaction log.

This means that before any events are recorded, Makeflow will emit the DAG it parsed to the top of the journal as shown in Figure 7.5. For each node in

the workflow, we record the node identifier and its original shell command. Next, we list any parents it may have, its source and target files, and the actual shell command used in the batch job. Because the DAG information is prefixed with #'s Makeflow will treat these lines as comments and safely ignore them when recovering a log.

With this information, it is possible for the profiling utilities to map the event records to actual task nodes in a single pass. Likewise, embedding the parsed DAG into the transaction log avoids the need to have to parse both the transaction journal and the DAG. Instead the profiling utilities only require the Makeflow log files to analyze, monitor, and profile the workflow.

2. **Record Workflow Status**: The transaction log only records event records. This means that the journal is only updated when a node is modified (i.e. changes state). It is not clear when the workflow started or stopped because the state of the workflow itself is not record. While it is possible to use some heuristics to determine if a workflow is complete (i.e. all tasks nodes are in the complete state), it would be difficult to determine if a workflow has failed or has been aborted.

   To make it easier for tools such as `makeflow_monitor` to determine the current execution status of a workflow, I simply emit workflow status records. Again, to avoid conflicting with the existing event record parser, I embedded this new status information as a comment. As demonstrated in Figure 7.5, when a workflow is started, a `# STARTED <TIMESTAMP>` record is emitted. When the workflow completes, `# COMPLETED <TIMESTAMP>` is recorded. For failed or aborted workflows, `# FAILED <TIMESTAMP>` or `# ABORTED <TIMESTAMP>` messages are recorded.

Having these workflow status records allows for the profiling tools to accurately assess the execution status of Makeflow.

3. **Include Debugging Symbols**: The final modification to the transaction log involves including debugging symbols generated by the compiler in the transaction log. With Weaver, the user can compile a workflow with the `-g` flag which will produce a DAG with embedded symbolic annotations as shown in Figure 7.6. In the example, there are two `Abstractions`, one that performs a `Map` using the `convert` application and another performing a `Map` using the `stat` program. For each task, there is a comment after the target and source file specification line denoting the symbol with which the task is associated.

   When the DAG is executed and it contains these symbolic annotations, this information will be propagated to the transaction journal as shown in Figure 7.5. After the node identifier and original command are emitted, the symbolic annotation is recorded. With this information tools such as `makeflow_report` can not only provide profiling information about individual tasks but also on `Functions` and `Abstractions`. This is important because with the previous version of the transaction log it would be difficult to map tasks to the higher-level workflow specification. By including debugging symbols in the DAG and in the transaction log, it is now possible to construct a complete picture of the workflow from the Python code to the Makeflow DAG to the transaction log.

```
xterm-color_32x32.jpg: /usr/share/pixmaps/xterm-color_32x32.xpm /usr/bin/convert
        # SYMBOL        Map[0](/usr/bin/convert {IN} {OUT},_Stash/0/0/0/0000001)
        /usr/bin/convert /usr/share/pixmaps/xterm-color_32x32.xpm xterm-color_32x32.jpg
xterm_32x32.jpg: /usr/share/pixmaps/xterm_32x32.xpm /usr/bin/convert
        # SYMBOL        Map[0](/usr/bin/convert {IN} {OUT},_Stash/0/0/0/0000001)
        /usr/bin/convert /usr/share/pixmaps/xterm_32x32.xpm xterm_32x32.jpg
parcellite.jpg: /usr/bin/convert /usr/share/pixmaps/parcellite.xpm
        # SYMBOL        Map[0](/usr/bin/convert {IN} {OUT},_Stash/0/0/0/0000001)
        /usr/bin/convert /usr/share/pixmaps/parcellite.xpm parcellite.jpg
xterm-color_48x48.jpg: /usr/share/pixmaps/xterm-color_48x48.xpm /usr/bin/convert
        # SYMBOL        Map[0](/usr/bin/convert {IN} {OUT},_Stash/0/0/0/0000001)
        /usr/bin/convert /usr/share/pixmaps/xterm-color_48x48.xpm xterm-color_48x48.jpg
xterm_48x48.jpg: /usr/bin/convert /usr/share/pixmaps/xterm_48x48.xpm
        # SYMBOL        Map[0](/usr/bin/convert {IN} {OUT},_Stash/0/0/0/0000001)
        /usr/bin/convert /usr/share/pixmaps/xterm_48x48.xpm xterm_48x48.jpg
dataset.py.stat: /usr/bin/stat /home/pbui/src/research/weaver/weaver/dataset.py
        # SYMBOL        Map[1](/usr/bin/stat {IN} > {OUT},_Stash/0/0/0/0000003)
        /usr/bin/stat /home/pbui/src/research/weaver/weaver/dataset.py > dataset.py.stat
abstraction.py.stat: /usr/bin/stat /home/pbui/src/research/weaver/weaver/abstraction.py
        # SYMBOL        Map[1](/usr/bin/stat {IN} > {OUT},_Stash/0/0/0/0000003)
        /usr/bin/stat /home/pbui/src/research/weaver/weaver/abstraction.py > abstraction.py.stat
compat.py.stat: /usr/bin/stat /home/pbui/src/research/weaver/weaver/compat.py
        # SYMBOL        Map[1](/usr/bin/stat {IN} > {OUT},_Stash/0/0/0/0000003)
        /usr/bin/stat /home/pbui/src/research/weaver/weaver/compat.py > compat.py.stat
options.py.stat: /home/pbui/src/research/weaver/weaver/options.py /usr/bin/stat
        # SYMBOL        Map[1](/usr/bin/stat {IN} > {OUT},_Stash/0/0/0/0000003)
        /usr/bin/stat /home/pbui/src/research/weaver/weaver/options.py > options.py.stat
```

Figure 7.6. Weaver Generated Debugging Symbols Example

When symbolic annotations are enabled, the compiler will record the `Function` *or* `Abstraction` *the task is associated with by embedding a comment before the task's shell command. This symbol is then used by the profiling utilities to group related tasks for analysis.*

The purpose of these augmentations to Makeflow is to facilitate and improve debugging and tracking of a user's workflow by enhancing the amount of provenance information in the transaction journal. This additional information is used by the toolchain's profiling utilities to analyze workflows, monitor their progress, and generate statistical reports about the components of the workflows.

7.4 Garbage Collection

The final modification to Makeflow involves using garbage collection to tackle the problem of intermediate files. As noted in previous chapters, in some scientific workflows, many intermediate files are generated throughout the course of the execution of the application. For small applications this does not pose a problem. However, for larger workflows too many intermediate files in the workflow sandbox can severely impact the performance of the overall application. Moreover, having too many intermediate files in the workspace not only adversely affects performance, but also makes it difficult for the user to browse their workspace and perform operations such as monitoring and debugging.

In Chapter 4, I described two techniques to address this problem. The first was the `Stash` structure which provides the user a convenient means of spreading their intermediate files across an internal directory hierarchy. The second method was to partition a workflow into smaller sub-workflows using hierarchical workflows. While both of these methods help prevent the pernicious problem of inode exhaustion, they do not handle the case where the user may have limited storage for their workspace. This could either be due to lack of storage capacity or system-enforced quotas. Additionally, these methods also do not address the case where the user only cares about the final outputs and not any of intermediate

ones. Instead of keeping the intermediate files around, it would be more desirable to remove them when the workflow manager has determined they are no longer necessary.

To address these types of situations, I augmented Makeflow to support automatic removal of volatile or temporary intermediate output files. Currently, users can schedule tasks to remove intermediate files as a part of the workflow, but such tasks violate the semantics of Makeflow and lead to strange behavior of the workflow is aborted and resumed. This is because Makeflow uses the existence of files (along with its journal log) to help it determine if a task has been completed. Removing a file will trigger the generating event to re-run if the Makeflow is resumed or restarted.

In the modified version of Makeflow, users can mark certain files as volatile or temporary. During execution of the workflow, Makeflow will periodically perform garbage collection on this list of temporary files, thus alleviating the user from having to schedule removal tasks. Since Makeflow understands that certain files are volatile it can be smarter about scheduling tasks that generate temporary files. For instance, the augmented Makeflow will avoid re-scheduling an already completed task if an input or output file is missing but was marked as a temporary file. However, if a re-scheduled failed or aborted task required input files that were previously garbage collected, then those parent tasks would also be re-scheduled. This way, the re-schedule task is guaranteed to have all of its necessary input files.

For this dissertation, I modified Makeflow to support the following methods of garbage collection:

1. **Immediately:** Temporary files are removed as soon as they are no longer needed. This is similar to reference counting [11], where once the number of

references to an object reaches zero it is then deleted. In my implementation, this technique is labeled `RefCount`.

2. **On-demand:** Temporary files are only removed when there is need for more space or inodes. This is used by many "stop-and-collect" garbage collectors [14] which will pause the program once it runs out of available resources and perform garbage collection to free up more. This method is appropriately called `OnDemand` in Makeflow.

3. **Incrementally:** In this method, Makeflow periodically removes a fixed amount of temporary files or only used a pre-determined amount of time to perform garbage collection. This is similar to incremental garbage collectors [7] used by applications that require real-time performance guarantees. There are two versions of this method: `IncrFile` which collects up to a user-defined amount of files (default is 16), and `IncrTime` which collects as many files as it can in a limited time window (default is 5 seconds).

Internally, reference counting is used to track which files are available for collecting. When the DAG is initially parsed, a list of volatile temporary files is stored in the `_MAKEFLOW_COLLECT_LIST` variable. This means that if users wish to mark a file as volatile, they simply need to append it to this variable:

```
out.0: in.0
    @_MAKEFLOW_COLLECT_LIST+=in.0
    command in.0 > out.0
```

Once the `_MAKEFLOW_COLLECT_LIST` variable is set, then Makeflow will create a collection table that maps each file in the list to the number of references it has in

the graph. To compute the amount of references, Makeflow traverses the DAG and increments the appropriate entry in the mapping table for each node that requires the volatile file. When a task is returned from the batch system, Makeflow checks if any of its input files are in the collection table. If it exists in the table, then the reference count in collection table is decremented. Furthermore, if the current garbage collection method is `RefCount` (i.e. reference counting), then the file will be removed immediately if the count reaches zero.

For the other methods (`OnDemand`, `IncrFile`, and `IncrTime`), Makeflow will periodically attempt to perform garbage collection after a certain amount of tasks have completed. Currently, this amount is limited to 5% of the total number of tasks. That is if a workflow has 100 tasks, Makeflow will attempt to perform garbage collection 5 times. This is done to space out the time between collection (otherwise all the other methods devolve into just reference counting). When a workflow successfully completes, a final attempt to collect files will be performed.

```
1 NFILES  = int(CurrentScript().arguments[0])
2 stat    = ParseFunction('stat {IN} > {OUT}')
3 files   = Iterate('ls > {OUT}', NFILES, '{i}')
4 stats_0 = Map(stat, files, '{BASE}.stat_0', collect=True)
5 stats_1 = Map(stat, files, '{BASE}.stat_1', collect=True)
6
7 Merge(stats_0, 'all.stats_0', collect=True)
8 Merge(stats_1, 'all.stats_1', collect=True)
```

Figure 7.7. Garbage Collection Benchmark Workflow

For this benchmark, we create a large amount of files by listing the directory and storing the contents into an output file. We then stat these files twice and merge these results into two tables. Only the final two tables are kept; all other intermediate files are marked for collection.

To evaluate these garbage collections, I constructed the benchmark in Figure 7.7 and executed on three different filesystems (local, AFS, and NFS) using Makeflow's local batch system. In this benchmark, I create a $10,000$ files by listing the directory and storing the output in a new file. I then perform `stat` twice on each of these files and then merge the results into two separate tables. To mark which input files to garbage collect, I set the `collect` keyword to `True` on lines 4, 5, 7, and 8. This tells the compiler to mark all of the inputs for these abstractions as volatile and allow the run-time system to remove them when they are no longer necessary. At the end of the workflow, then, I should only have the two resulting tables as all of the intermediate output files are marked as temporary and volatile.

For each filesystem, I executed the workflow multiple times with all four of the garbage collection methods described above and with garbage collection turned off as a baseline. The execution times for these benchmarks is shown in Figure 7.8. For the local filesystem, garbage collection lead to a minimal amount performance increase, while on NFS, the methods yielded a slight performance decrease. On AFS, however, garbage collection had a great impact on the workflows' execution times. Here, `RefCount` lead to a severe performance loss compared to not using any garbage collection, while `IncrTime` and `OnDemand` lead to significant performance increases. `IncrFile` only produced a slight performance increase.

The differences in the performance impacts of the garbage collection methods on these filesystems can be in part explained by the information in Figures 7.9 and 7.10. The first chart shows the percentage of the workflow's execution time that removing files occupied. For the local filesystem, deleting volatile files is a relative inexpensive operation and thus the garbage collection methods did not take up

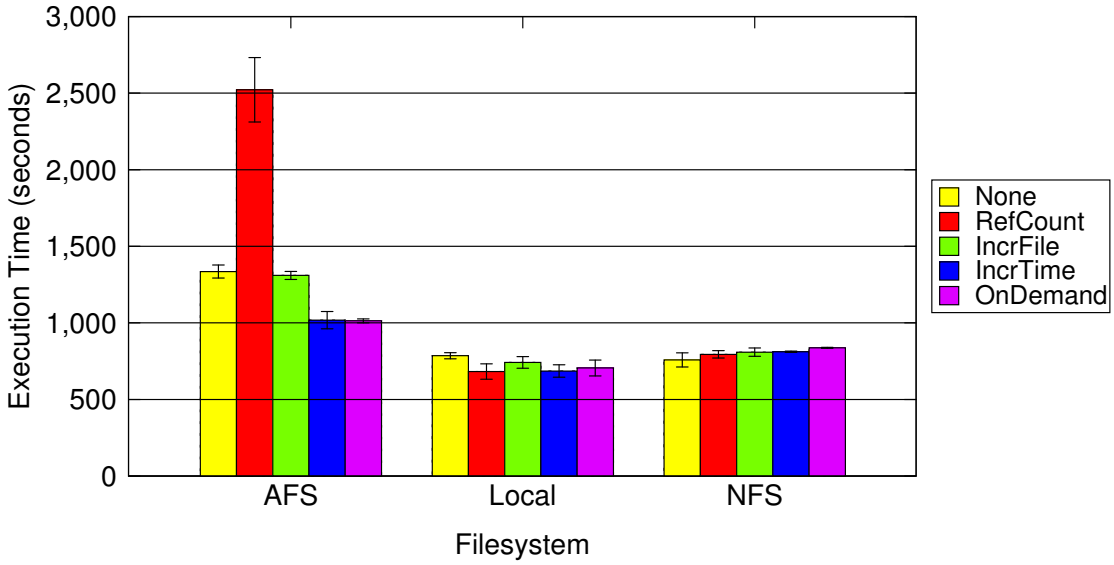


Figure 7.8. Garbage Collection Benchmark Execution Times

This graph presents the mean execution of the benchmark workflow using different garbage collection methods (none, reference counting, incremental by file, incremental by time, and on demand) across multiple filesystems (AFS, Local, NFS). On the Local filesystem, garbage collection provided a modest performance increase, while on NFS it lead to a slight performance decrease. On AFS, collecting garbage lead to dramatically different results, depending on the method utilized.

much of the overall execution time. On NFS, however, due to the overhead of the network RPCs that must take place, removing files proved to be a somewhat costly and thus slightly degraded performance. With AFS, the garbage collection techniques also occupied a larger percentage of the workflow's execution time. In particular, the `RefCount` and `IncrTime` methods took a much larger percentage of the running time and had the worse performances. The cost of removing files on AFS, however, was offset by the gains in getting to list a smaller directory in the case of `IncrTime` and `OnDemand`. That is, while deleting files is costly in AFS, so is listing the directory, especially if it is full of files.

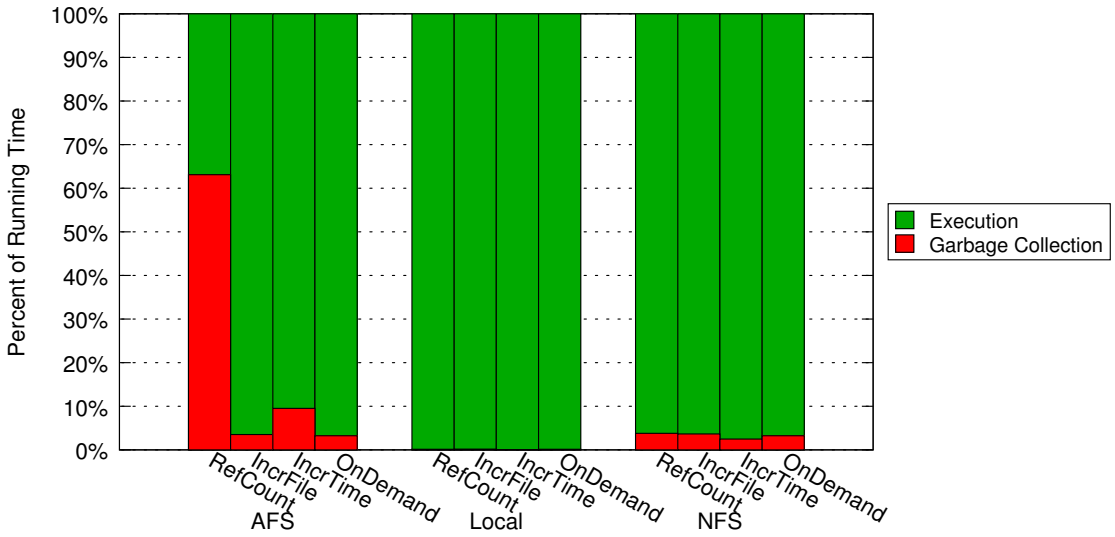


Figure 7.9. Garbage Collection Benchmark Running Time Percentages

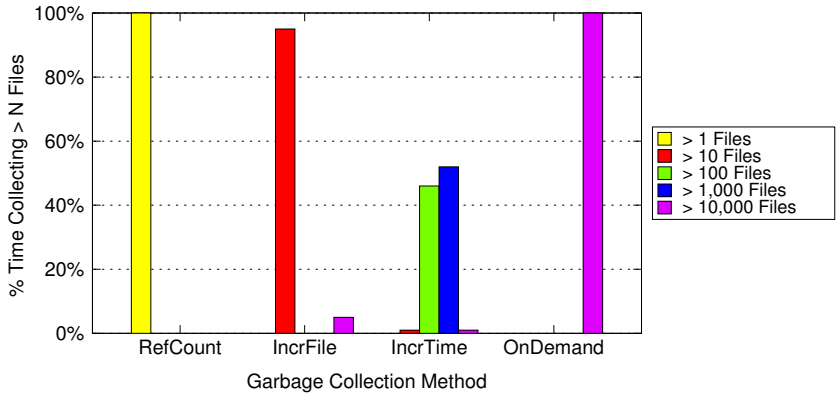
This charrt hows the percentage of the workflows' running time garbage collection occupied. On AFS, removing files proved costly and thus ate up a signficant amount of execution time. For the local filesystem, deletion was much more inexpensive and thus did not significantly impact the execution time. On NFS, deletion was also costly and lead to slight decrease in workflow performance.

Figure 7.10 provides further insight into the performance impacts of the garbage collection methods. These histograms show how often each method collected $1-10$ files, $10-100$ files, $100-1,000$ files, $1,000-10,000$ files, and more than $10,000$ files during each cycle. Unsurprisingly, the `RefCount` method shows 100% frequency of deleting one file across each filesystem. This in part explains why `RefCount` is so expensive on AFS; for each volatile file, it will perform a single unlink command. At the other end, it is also unsurprising that `OnDemand` method always collected more than $10,000$ files at a time. This is because workflows with this method only triggered a collection cycle when the number of inodes surpassed the limit of $2^{14} = 16384$ files. Likewise, `IncrFile` usually collected 16 files per cycle because that is what the method was designed to do. However, this collection amount did

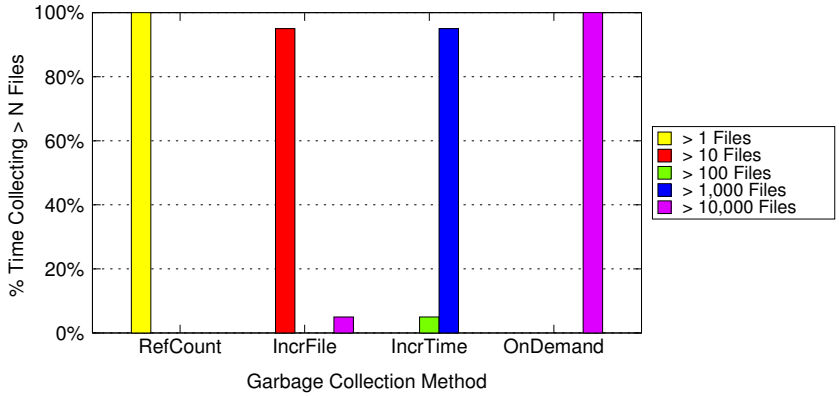
not keep up with the number of volatile files because the `IncrFile` technique also had to spend a modest amount of its collection time removing more than $10,000$ files. Out of all the methods, `IncrTime` displayed the most variation. On AFS, this method spent nearly equal time deleting $100 - 1,000$ and $1,000 - 10,000$ files at a time. This explains why its performance on AFS was as good as `OnDemand`; it was able to delete large batches of files at once, and thus minimized the amount of network callbacks and drove down the cost of listing a directory. For local and NFS, `IncrTime` spent most of it's time deleting $1,000 - 10,000$ files per cycle.

All of these results demonstrate the performance impact of these garbage collection techniques. Overall, the `IncrTime` and `OnDemand` techniques appear to work the best, particularly on AFS, because they delete a large batch of files at once, rather than a few at a time. `IncrTime` could be brought closer to the performance of these two methods by simply increasing its threshold (the benchmarks utilized the default of 16). Reference counting, on the other hand, is not recommend, especially for AFS, where it led to a significant performance decrease.

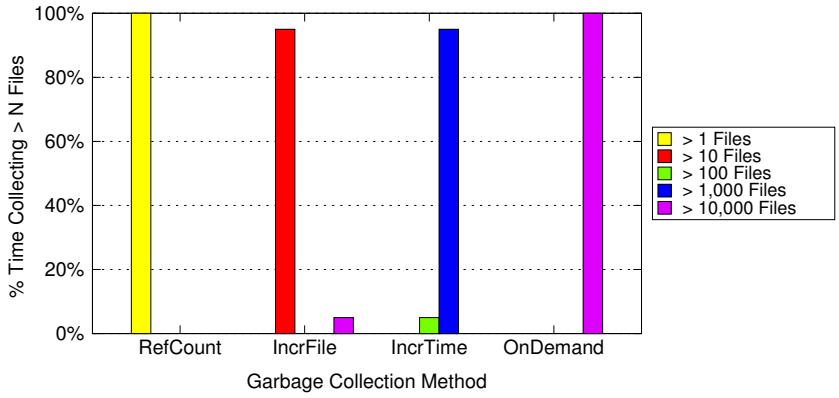
As noted previously, the compiler toolchain utilizes Makeflow as its target runtime system. While the workflow manager is provides a stable and easy target for generating workflow DAGs and executing them on a variety of distributed platforms, it was insufficient for some of the features desired by the toolchain. Because of this, I modified Makeflow to support local task variables, nested Makeflows as first-class objects, additional provenance and symbolic annotations, and garbage collection. All of these additions are utilized and taken advantage of by various parts of the compiler toolchain and serve to enhance Makeflow as an capable workflow manager for distributed data intensive scientific workflows.



(a) AFS Histogram



(b) Local Histogram



(c) NFS Histogram

Figure 7.10. Garbage Collection Benchmark Collection Histograms

These histograms show the percentage frequency of how many files were deleted during each collection cycle. For instance, `RefCount` *always removes one file at time, so workflows using that method will have* 100% *frequency of deleting* $1 - 10$ *files.*

CHAPTER 8

CONCLUSION

With the growing amount of computational resources available to researchers today and the explosion of scientific data in modern research, it is imperative that scientists be able to efficiently and effectively construct data intensive applications that harness these vast computing systems. To address this need, I proposed applying concepts from traditional compilers, linkers, and profilers to the construction of distributed workflows and evaluated this idea by implementing a complete compiler toolchain that allows both novice and expert users to compose distributed data intensive scientific workflows in a high-level programming language.

8.1 Automatic Optimizations

In Chapters 4, I described a few compiler optimization techniques that can be utilized by users to improve the performance their workflows. While these techniques demonstrated significant performance improvements in certain situations, they all required some form of manual user intervention or input. For instance, instruction selection requires the user to enable the `native` keyword argument, while task inlining requires the user to specify a particular `group_size`. Likewise, hierarchical workflows requires the user to manually partition the workflow into

separate components. In all of these cases, the compiler cannot make a good decision by itself since it does not have enough information to make an accurate decision on how to apply these optimizations.

One possible way to enable the compiler to perform automatic optimizations is to leverage the additional provenance information described in Chapter 7 to implement profile-guided optimization [92]. In this new technique, the compiler enables symbolic annotations and samples the workflow by executing small subsections of the workflow with different optimization parameters and examine the transaction journal. Because of the annotations, the compiler can parse the log and map the execution times back to the high-level `Abstractions` and `Functions` and determine which optimization parameters yielded the best performance and thus optimize the whole workflow accordingly.

Another method would again take advantage of the provenance information, but this time to mine a large corpus of transaction journals for patterns. Instead of having the compiler perform sampling, the user would apply a new data-mining tool to a set of annotated transaction logs to extract any performance patterns from previous set of executions. Previous work has used data-mining for debugging purposes [32], but not necessarily for optimizing a workflow. In some ways, this would be similar to what a tracing just-in-time (JIT) [46] compiler does, but offline instead of during execution.

8.2 Dynamic Workflows

Another major possible consideration for the compiler is to support dynamic workflows. Currently, the compiler can only generate static workflows where all the tasks must be enumerated before execution. While this encompasses a large

number of scientific data processing applications, it does not allow for iterative applications such as optimization, filtering, and simulation where run-time decisions that affect the direction of the workflow must be made. This restriction is primarily due to the fact that the run-time system, Makeflow, does not provide the necessary primitives for dynamic scheduling.

Despite this limitation, it still is possible to do some forms of dynamic workflows with the toolchain. For instance, the one way of constructing an iterative application using the is to have a top-level script that simply compiles and executes a `Nest` for each iteration. Because the Weaver compiler is written as a Python library with a front-end script, it is possible for normal Python scripts to programmatically construct, compile, and execute workflows. While this would mostly work, the problem with this technique, however, is that the user does not have the ability to restart or resume their workflow unless they manually implement some form of checkpointing.

Figure 8.1 provides a contrived example of this type of application. In this demonstration, we construct a `Nest` whose sole task is to create a file `.dat` file. After each execution of the `Nest` we check for how many `.dat` files we have. If we have two, then we break out of the loop. One can imagine, how this example can be modified to perform an optimization or simulation type of application.

Another way to implement dynamic workflows is to use a system of *trampolining*. Normally this method is used to used in dynamic programming languages such as Scheme to support continuations [47] by having an outer function call an inner function to generate the next function to call [6]. In the context of DAG-based workflows, we would basically have a rule that generates another workflow and then executes that generated DAG. As can be imagined, this can be quite

```
from weaver.function    import ParseFunction
from weaver.nest        import Nest
import glob

touch = ParseFunction('touch {OUT}')

for i in range(5):
    with Nest() as nest:
        touch(outputs='{0}.dat'.format(i))
        nest.compile()
        nest.execute()

    if len(glob.glob('*.dat')) == 2:
        break
```

Figure 8.1. Iteration Workflow Example

This provides an example of how to construct an iterative application such as optimization or simulation by using the Weaver compiler library directly in a normal Python script to construct a `Nest`*, compile it, and then execute it.*

complex and difficult to manage manually, but it is a viable technique and would make applications such as filtering possible.

In order to fully support dynamic workflows, Makeflow must be modified to provide a means of dynamic task generation. If the run-time system allowed tasks to generate other tasks, then the toolchain could be updated to support a broader range of workflow applications. For instance, Swift [120, 126] is implemented as a compiler and yet it supports dynamic workflows with conditionals (e.g. `if` and `switch` statements) due to its run-time system. Likewise, Skywriting [82] is an interpreter-based functional language for developing cloud applications that supports iteration and recursion due to the fact that its CIEL [83] execution engine is capable of dynamically constructing DAGs. Augmenting Makeflow to support features for defining tasks at run-time would enable dynamic applications such as optimization, filtering, and simulations.

8.3 Compiling to DAGs

The discussion of dynamic versus static workflows often brings into question why a compiler and not an interpreter? For the most part, this is a poorly framed question because even traditional interpreters for languages such as Python and Lua compile the input source into byte-codes that are processed by an internal virtual machine. In other words, it is unclear what the difference between a compiler and an interpreter are suppose to be since many interpreters do compilation internally (they just do it at run-time). As demonstrated earlier, we can simulate a basic interpreter by constructing, compiling, and executing workflows on-demand within a normal Python script.

I believe that the more interesting and relevant question is why compile a workflow to a DAG? That is why produce DAGs for Makeflow, when the toolchain could dispatch jobs as it parses the workflow specification on its own? The initial answer is due to practicality. Makeflow exists and does a great job of executing workflows in a portable and reliable way and so to target it I must generate a DAG. If I were to remove the dependency on Makeflow, I would basically have to re-create a workflow manager inside the compiler to make it a true interpreter, duplicating previous work instead of researching new areas. Despite this, however, given enough time, I could in fact re-implement Makeflow inside of Weaver to create an intrepreter, and so practicality is not a sufficient answer.

A more convincing reply is that compiling a DAG allows us to apply optimizations. When we compile a workflow, we have the opportunity to examine the structure of the workflow and modify it to improve its performance. A prime example of this is inlining tasks as presented in Chapter 4. With this optimization, we can group or cluster elements of a dataset together and have them all processed

by a single worker to minimize dispatch latency. Unfortunately, this reason also falls short, because there is nothing that stops an interpreter from also performing most of these optimizations on the fly as done by traditional programming languages such as Python.

The best reason for using a DAG as an intermediate representation of the workflow is that it allows the compiler and run-time to analyze the entire workflow and make intelligent decisions. More specifically, having a complete representation of the workflow as a DAG means that the toolchain and run-time can perform whole program optimizations [68], dynamic scheduling [95], and resource allocation [103]. The first feature refers to the fact that while an interpreter can perform optimizations on individual `Abstractions` or `Functions`, it cannot perform `Nest` or workflow level transformations without constructing a DAG. The second ability means that the run-time system can analyze the whole structure of the graph and re-adjust how it schedules its tasks. The third capability is similar to the second except it means that the workflow manager can analyze the DAG to optimize resource allocation. A hint of this was presented in Chapter 7 where we used the `dag_width` function to proportionately allocate local batch jobs for nested Makeflows. These types of features are only available if we have a complete graph of the workflow.

8.4 Beyond Distributed Computing

Looking beyond distributed computing, I believe that software engineering approach underlying by my dissertation is applicable to many software construction problems today. While is important to be able to create highly-optimized code, most programmers develop software by piecing together these specialized com-

ponents to form sophisticated applications. When using the toolchain presented here, users are basically coordinating an *ensemble* of applications and abstractions to form a distributed scientific workflow. What is important to note is that the toolchain is not designed for the users creating the specialized applications, but the developers who will be gluing these components together to synthesize a larger application. This is the software engineering concept in my work:

Abstract the performance critical components into individual independent modules which users can combine to form more sophisticated applications.

Outside the distributed computing field, many programming language, computer architecture, and software engineering researchers are tackling the problem of how to enable more users the ability to harness the computation power of high-performance parallel devices such as multi-core machines and graphics processors. Some researchers promote low level mechanism such as threads [24] or MPI [39], while others call for new languages such as Chapel [27] or Fortress [108]. Still more push for frameworks such a OpenCL [81].

As my dissertation demonstrates, another approach that should be considered is to focus on creating a set of libraries and utilities that programmers can combine to create applications that take advantage of multi-core machines and GPUs. For instance, a library of data structures optimized for multi-core would enable users to substitute their existing data structure implementations with these high performance modules. Likewise, a library of mathematical functions that take advantage of the GPU could be used by a variety of data processing applications to greatly improve their throughput.

Having these optimized components, however, is not enough. Users must be able to combine them with their existing code and with other specialized components. Whether it is through a domain-specific language, a library API, a set of executables, or any other method, these optimized components need to be available and allow users to compose them together to create new high-performance applications. A compiler toolchain, as demonstrated by my dissertation, is an effective tool for integrating these optimized modules and allowing users to construct sophisticated applications that take full advantage of available computational resources.

8.5 Impact

Recalling the Fred Brooks' idea of computer scientist as toolsmith, the success of any systems software research is measured by the collaborations it enables and the research it promotes. Under these terms, the compiler toolchain is a success. Elements of the toolchain have been in use for the past for years. In additional to a few external users, many current users of the toolchain are collaborators here at the University of Notre Dame.

For instance, it is currently employed in various operations for the Computer Research Vision Laboratory (CVRL) at Notre Dame. Specifically, it is used to perform biometrics experiments and as the transcoding framework that processes terabytes of data for the BXGrid [17, 20] web portal. Likewise, the toolchain has been adopted by members of the Notre Dame Biological Computing group to port existing genomic workflows to a variety of distributed systems [67, 114]. Using the toolchain, these researchers are now able to run previously infeasible workflows to analyze genomic data on campus distributed systems with much shorter

turn-around time. Thus, the toolchain allows the researchers to concentrate on analyzing gene sequences and discovering new biological breakthroughs.

As such, the compiler toolchain is succeeding in its goal of enabling novice and expert users to efficiently and effectively construct distributed data intensive scientific workflows.

APPENDIX A

WEAVER API

Chapter 3 presented an overview of the programming interface and execution model of the Weaver workflow compiler. Recall that Weaver provides a simple, though restricted, programming model that consists of **datasets**, **functions**, **abstractions**, and **nests**. These concepts are the fundamental building blocks of the Weaver application programming interface (API) and are implemented as a custom Python package consisting of modules, classes, and functions that end users combine and extend to define their distributed scientific workflows.

This appendix provides an in-depth examination of the programming model the compiler presents to end users along with example Python source code.

A.1 Datasets

Many scientific computing tasks involve processing a repository or collection of experimental data, which are normally stored as files on a physical filesystem. In the Weaver programming model, collections of data objects are organized into **datasets**, where each object's string (i.e. `__str__`) method returns the location of the file that contains the data. This simple convention allows for datasets to take the form of a Python list, set, generator function, or any other Python object that implements the language's native iteration protocol.

Although specifying a dataset can be as straightforward as defining a list of file paths, Weaver provides a collection of custom `Dataset` objects that simplify the specification and selection of input data. Each object in these `Dataset` collections contains the path to the data file as required by the programming model, along with a set of attributes shared by all the members of the dataset. This common set of metadata properties can be accessed and manipulated by the user through the `Query` function, which will be explained shortly.

One example of a `Dataset` provided by the Weaver framework is the `Files Dataset`. Since the most common type of dataset is simply a group of data files, Weaver provides the `Glob` constructor, which given a file path pattern, this dataset builder will return the set of file objects that match the specified pattern. Another common method for keeping track of files is to store the list of data paths in a text file. Weaver provides a simple `FileList` object for constructing this type of collection. Each object in both of these collections contains the location of the file, along with relevant filesystem metadata of each file such as size and timestamps. An example of a `Files Dataset` generated by the `Glob` constructor is shown below in Figure A.1.

In addition to files stored on a filesystem, another common source for scientific data is a SQL [28] database. For these datasets, Weaver provides a simplified database specification and querying interface that facilitates accessing information stored in conventional SQL databases such as MySQL [84] or SQLite [90]. Besides specifying the details about how to connect to the database as shown in Figure A.1, the user only needs to define a `path` function which determines the location of the data file based on the object record returned by a SQL query. The user may either directly map the database record to a file on disk, or the user may

```
# Define dataset using Glob constructor
fls_ds = Glob('/path/to/files/*.txt')

# Filter files dataset for sizes > 1024
my_fds = Query(fls_ds, fls_ds.c.size > 1024)

# Define dataset using MySQL constructor
sql_ds = MySQLDataset('localhost', 'biometrics', 'files')

# Filter SQL records based on eye color
my_sds = Query(sql_ds,
            Or(sql_ds.c.EyeColor == 'Blue',
               sql_ds.c.EyeColor == 'Green'))
```

Figure A.1. Weaver Dataset Examples

This figure shows examples of defining `Datasets` *using a couple of provided Weaver constructors and demonstrations on how to filter these collections based on the metadata associated with each item in the dataset. Note that while the first dataset is a list of files and the second one is a MySQL database, both can be manipulated using the same ORM interface provided by Weaver's* `Query` *function.*

materialize a file containing information from the database record and return the path to that generated file. In either case, it is up to the user to specify how to translate the database record to a physical file location as required by the Weaver programming model. An example of mapping a database record to a physical path for data from the BXGrid [17] repository is provided in Figure A.2. In this demonstration, the `fileid` of the database record is used to map into the filesystem namespace provided by the Parrot [112] BXGrid driver [19].

Sometimes it is necessary to filter or select a subset of the dataset from a larger collection before processing it. For instance, a scientific database may contain thousands of records, but the user is only interested in a specific subset for experimentation. To facilitate this selection operation, Weaver provides a `Query` function that allows the user to filter items in Weaver `Datasets`. This selection

```
# Function to map database record to physical file location
def bxgrid_path(self, object):
    return '/bxgrid/{0}/fileid/{1}'.format(
        self.db_host, object['fileid'])

# Configure BXGrid database dataset
dataset = MySQLDataset(host='localhost', name='biometrics',
            table='files', path=bxgrid_path)

# Filter dataset based on fileid and size
files   = Query(dataset,
              Or(And(dataset.c.fileid == '321',
                     dataset.c.size > 1040000),
                 And(dataset.c.fileid == '322',
                     dataset.c.size > 1040000)))
Map('stat {IN} > {OUT}', files, '{BASE}.stat0')

# Select data without fileids
files   = Query(dataset,
                dataset.c.fileid == None, limit=10)
Map('stat {IN} > {OUT}', files, '{BASE}.stat1')

# Select data with fileids beginning with '12'
files   = Query(dataset,
                dataset.c.fileid % '12%', limit=10)
Map('stat {IN} > {OUT}', files, '{BASE}.stat2')

# Select data with specified extensions
files   = Query(dataset,
                dataset.c.extension | ('gz', 'abs.gz'), limit=10)
Map('stat {IN} > {OUT}', files, '{BASE}.stat3')
```

Figure A.2. BXGrid SQL Dataset Examples

This code listing demonstrates how to setup a connection to the BXGrid repository and provides examples of querying the database for biometric data. The first few lines define `bxgrid_path`*, which is used to determine how to map the database record returned by the queries into a physical file location. The ORM language provided by Weaver's* `Query` *function is similar to the expression language made popular by SQLAlchemy and closely resembles standard SQL.*

is possible because objects in Weaver `Dataset` collections contain metadata information common to each item in the set, and thus allow for the user to filter these sets of objects based on their attributes.

The Weaver `Query` function exposes this selection mechanism by supporting a SQL-like query expression language which translates the user-defined queries into an appropriate form for the underlying data structure. This query expression language is similar to the SQLAlchemy expression language [105], a popular Python object-relational mapping (ORM) [62] system. For datasets that are actual databases, the function will translate the ORM query expression into the appropriate SQL expression and use the generated SQL to perform the query on the database server. In the case of datasets that are collections of Python objects, the ORM query expressions are translated into a series of filter functions that are applied to each object in the dataset to produce the desired subset of the data collection. To use the `Query` function, the user simply specifies the name of the dataset, followed by an ORM query expression. As shown in Figures A.1 and A.2, this allows users to filter their datasets in a simple and consistent manner.

It is important to note that these filtering and selection operations occur during compilation. In fact, all of the `Dataset` collections are enumerated at compile time because Weaver is a *static* workflow compiler. This means that all elements of the DAG, including input and output datasets, must be generated and enumerated by Weaver during compilation and thus before executing the actual workflow. Moreover, this also denotes that the structure of the workflow is static and does not change over the course of the workflow execution.

In summary, users may specify **datasets** using normal Python collections such as lists, tuples, or sets, or they may use one of the provided Weaver `Dataset` constructors such as `Glob`, `FileList`, `SQLDataset`. Utilizing one of the Weaver `Dataset` constructors further enables selection and filtering using a simple ORM system provided through the `Query` function.

A.2 Functions

The second major component of most scientific workflows are the executables used to process the data. The Weaver programming model accounts for these executables by providing the notion of a **function** specification object. In Weaver, a `Function` is a Python object that defines the interface to an external application or embedded script. It specifies information such as the path to the executable and how arguments to the `Function` are to be formatted to generated a *shell command* that can be executed to perform the desired operation. Like objects in a **dataset**, each `Function` also corresponds to a physical file, in this case an executable or script, on the filesystem.

As with `Datasets`, Weaver provides a set of custom Python components designed to expedite and simplify the specification of workflow functions. The base constructor is the generic `Function` class that contains the path to the executable as well as a couple of methods: `command_format` and `__call__`. The first method specifies how to generate the appropriate shell command string needed to execute the task given a set of input and output files, while the second method allows the `Function` object to be called as a normal Python function. The side-effect of executing a `Function` is scheduling a task generated based on the `command_format` method and the arguments to the `__call__` method. Examples of defining and calling various Weaver `Functions` are demonstrated in the source code examples in Figure A.3.

The *shell command* generated by a `Function's command_format` method is internally based on the object's `cmd_format` attribute which can be specified during construction as shown in the `Convert` example in Figure A.3. This attribute is a string template that is used to substitute in values using the rules specified

```
# 1. Create convert function directly using Function constructor
Convert = Function('convert', cmd_format='{EXE} {IN} {OUT}')
Convert('example.jpg', 'example.png')

# 2. Create cat function using utility ParseFunction constructor
Cat = ParseFunction('cat {inputs} > {outputs}')
Cat('/etc/hosts', 'hosts.txt')

# 3. Create script function using ShellFunction constructor
Touch_SH = ShellFunction('touch $2 && chmod $1 $2',
                         cmd_format='{EXE} {ARG} {OUT}')
Touch_SH(outputs='touch.txt', arguments='600')

# 4. Create sum function from inline Python code
def py_sum(*args):
    print(sum(map(int, args)))
Sum = PythonFunction(py_sum)
Sum(outputs='sum.txt', arguments=[0, 1, 2, 3, 4, 5])

# 5. Create meta-Function using Pipeline constructor
GetPids = Pipeline(["ps aux",
                    "grep {ARG}",
                    "awk '{{print $$2}}' > {OUT}"], separator='|')
GetPids(outputs='makeflow.pids', arguments='makeflow')
```

Figure A.3. Weaver Function Examples

The first example shows how to specify a `Function` *directly by using the base* `Function` *constructor and how to call the constructed object in order to schedule a new task. The second example uses the* `ParseFunction` *utility function to specify the executable and* `cmd_format` *in a much simpler manner. The third example demonstrates how to create a* `ShellFunction` *which is an embedded shell script that will be materialized by the compiler for the user. The fourth example is similar, except this time an inline Python function is used instead of an embedded script. The final example demonstrates how to combine multiple* `Functions` *into a meta-*`Function` *using the* `Pipeline` *constructor.*

in Table A.1. Looking at the `Convert` example in Figure A.3, when the __call__ method is called internally after executing the line:

```
Convert('example.jpg', 'example.png')
```

The object's `command_format` method would yield the following based on the

arguments to `__call__` and the internal `cmd_format` attribute:

```
/usr/bin/convert example.jpg example.png
```

The goal behind all of this formatting and templating is to provide the end user with a simple but flexible way of specifying how the `shell command` for each function should be constructed.

TABLE A.1

WEAVER COMMAND FORMAT TEMPLATE

Key	Alias	Substitution Value
{executable}	{EXE}	Full path to the executable associated with `Function`.
{arguments}	{ARG}	Command line arguments to `Function`.
{inputs}	{IN}	Input file arguments to `Function`.
{outputs}	{OUT}	Output file arguments to `Function`.

Because specifying an executable and the `cmd_format` is a relatively common operation, Weaver includes a `ParseFunction` utility function that will automatically produce the correct type of function for the user. For instance, in Figure A.3 the `Cat Function` is constructed by call:

```
Cat = ParseFunction('cat {inputs} > {outputs}')
```

In this example, the `ParseFunction` takes the input string and constructs a `Function` where the executable (e.g. `cat`) is the first token in the input string and the `cmd_format` is the remainder of the input (e.g. {`inputs`} > {`outputs`}).

When calling a `Function`, the user may specify `inputs`, `outputs`, `arguments`, `includes`, `local`, and `environment` as keyword arguments to the `__call__` method. These parameters are used in following manner:

- `inputs`: This parameter specifies the input files to be processed by the `Function`. Normally the parameter would be `Dataset`, but if it is not, then all of the items in the collection are converted to `Files` using Weaver's `Makefile` utility function which calls each object's `__str__` method to determine the path to the object. If the `Function` does not expect any inputs, then this parameter may be omitted.

- `outputs`: This parameter specifies the output files the `Function` should create. If no outputs are to be generated, then this parameter may be omitted. The `outputs` parameter may either be a list of output files or a template string. In the case of a string template, the output list is generated by iterating over the list of `inputs` and substituting elements in the string template with values generated according to the rules in Table A.2.

- `arguments`: This parameter specifies any non-file command-line arguments to include in the *shell command*.

- `includes`: If there are any files that need to be included with the `Function` (e.g. libraries, configuration files, etc.), they can be specified by the user by setting this parameter.

- `local`: The scheduled task generated by the `Function` can be forced to use the `local` batch system by setting this parameter to `True`.
- `environment`: The user may pass a Python `dict()` to this parameter to specify any environment variables that should be set for the task scheduled by invoking this `Function`. Note, that this only sets the environment variables; it does not export them. Chapter 7 has more details on how environmental variables work.

TABLE A.2

WEAVER OUTPUTS TEMPLATE

Key	Alias	Substitution Value
{fullpath}	{FULL}	Full path of the corresponding input file.
{fullpath_woext}	{FULL_WOEXT}	Full path of the corresponding input file without extension.
{basename}	{BASE}	Base name of the corresponding input file.
{basename_woext}	{BASE_WOEXT}	Base name of the corresponding input file without extension.
{i}	{NUMBER}	Current index of corresponding input file in hexadecimal.
{stash}		Path of the next available `Stash` file.

In addition to the `ParseFunction` utility function, Weaver also includes two `ScriptFunction` constructors: `ShellFunction` and `PythonFunction`. Normally, `Functions` refer to external executables such as those typically found in the system's `/usr/bin` directory or locally in the user's workspace. Sometimes, however, it would be convenient to embed a short script or function in the DAG specification rather than actually creating a script in the filesystem *a priori*. For these situations, Weaver supports `ScriptFunctions` which are `Functions` that will create the actual executable on-demand from source embedded in the DAG specification.

The first type of `ScriptFunction` is a `ShellFunction`. As the name suggests, this constructor can be used to specify a short shell script from a string stored in the workflow script. In Figure A.3, the third example demonstrates how to construct a `ShellFunction` that creates a file using `touch` and sets the permissions on the file using `chmod`. Once constructured, the `ShellFunction` behaves as any other `Function` and may be called to generate a task. By default the `/bin/sh` shell command is utilized as the interpreter, but the user may change this by setting the `shell` keyword argument to another value (e.g. `/bin/bash`).

The second type of `ScriptFunction` is a `PythonFunction`, which can be used to convert an inlined Python function into an external executable on-the-fly. Figure A.3 provides an example of creating a `PythonFunction` that converts the `py_sum` function that prints the sum of its arguments into a `Function`. As with the `ShellFunction`, once defined, the `PythonFunction` acts as a normal `Function`. It is important to the note that when using a `PythonFunction`, the user must take care to ensure that source Python function imports any external modules it requires inside the function definition. This is because the Python function is extracted from its original source and written to a new script file which may lack

the same module namespace in which it was defined. For convenience, the `os` and `sys` are automatically imported in the generated `PythonFunction` and are thus available to the source Python function.

Finally, Weaver allows users to combine multiple `Functions` into a single meta-`Function` by using the `Pipeline` utility function. This is demonstrated in the fifth example in Figure A.3. In this example, we combine three `Functions` using standard UNIX pipe `"|"` syntax to find the process IDs of any instances of `makeflow`. As with all the other constructors, `Functions` created using the `Pipeline` utility behave like any other `Function`. There are a few reasons to use `Pipeline`:

1. **Dependencies:** By using `Pipeline` to combine multiple `Functions`, the scheduled task is guaranteed to have all of the necessary dependencies. If done as normal `Function`, then each of the executables would have to be added as a dependency to the `Function` manually. Using the `Pipeline` utility allows for automatic detection of the executable and thus dependency.

2. **Ordering:** Using a `Pipeline` enables the user to perform a specific order of operations without the use of file dependencies. For instance, it may be necessary to create a file and then modify it as in the case of the third example in Figure A.3. Although the example uses a `ShellFunction` instead of a `Pipeline` the idea remains the same. If this `ShellFunction` were split into two separate tasks, then it is possible for a race condition to occur.

3. **Optimization:** By combining a sequence of operations into a single meta-`Function` and thus reducing the amount of task dispatching required to perform the desired computation, the `Pipeline` function serves as way to perform manual function inlining [30] or vertical clustering [63].

4. **Clarity:** The final reason to use the `Pipeline` function is that the syntax makes it clear that user intends to connect all of these functions together.

Altogether, the `ParseFunction`, `ShellFunction`, `PythonFunction`, and `Pipeline` constructors provided by Weaver allow end users to specify their `Functions` in a straightforward and flexible manner.

A.3 Abstractions

The third component in the Weaver programming model is support for **abstractions**, which are patterns or models of computation with a precise set of semantics. As with datasets and functions, Weaver provides a built-in collection of distributed computing `Abstractions` to the end user as higher order functions that the user explicitly invokes in order to utilize the pattern in a workflow.

The first `Abstraction` is `AllPairs`. This is an abstraction that is frequently used in fields such as biometrics and data-mining [78]. In this pattern of computation each member of one dataset is compared to each member of a second dataset to produce a matrix that contains the resulting scores for each comparison. Because each individual comparison can be executed independently of each other, the workflow tasks for this abstraction can be scheduled to run in parallel.

The second `Abstraction` is `Map`, which is a common pattern used for work that exhibits data parallelism. `Map` takes an input function and applies it to each item in the input dataset. The results of each function application is stored in a collection of output data objects. Since each function application is independent of each other, the individual tasks in this pattern can be executed concurrently. Weaver also provides a related `Abstraction` called `Iterate`, which is just `Map`, except the input list is not a set of files, but arguments to passed to the `Function`.

The third `Abstraction` provided by Weaver is `MapReduce`, which is a distributed computing abstraction first introduced by Google [35] and made popular by Hadoop [51] for large scale data processing and analysis. In this pattern, a `mapper` function is applied to the initial set of inputs to generate a group of intermediate output files which are partitioned, sorted, and then passed to the `reducer` function for aggregation. All the tasks in the both the `mapper` and `reducer` phases exhibit data independence and therefore can be run in parallel.

The final `Abstraction` is `Merge`. This pattern of computation involves performing a parallel reduction [64] of the input dataset to merge or concatenate the original data into a single output. In the original version of the compiler [21], this `Abstraction` was an internal function that was only called if the user requested for the outputs of an `Abstraction` to be aggregated into a single file. In the most recent version of the compiler, this `Abstraction` is now exposed directly to the user and must be called explicitly to merge output files into a single target.

In Weaver, an `Abstraction` is conceptually a `Dataset` that is constructed by applying one or more `Functions` to an set of inputs to produce a set of outputs by performing a specific pattern of computation. These generated output files are yielded when the `Abstraction` is iterated over by another object. That is whenever the results of an `Abstraction` are required, for instance by another `Abstraction` or by a `Function`, the `Dataset` generated by executing the `Abstraction` is supplied. As a side-effect of determining this output `Dataset`, the `Abstraction` schedules an appropriate set of tasks to actually perform the computation in the generated workflow.

Because Weaver `Datasets` and `Functions` are designed to behave as if they are normal Python iterators and functions, implementing these patterns of com-

putations as `Abstractions` is rather straightforward. For instance, a simplified version of the `Map Abstraction` can be defined as follows:

```
for i, o in zip(inputs, outputs):
    yield function(i, o)
```

where `inputs`, `outputs`, and `function` are the arguments passed to the `Map Abstraction`.

Figure A.4 demonstrates how these `Abstractions` are invoked and how to connect the outputs of one `Abstraction` to another. In the first section of the listing, a `MapReduce` is performed to determine the word count of the Weaver source code. The output of this operation, which is a collection of files, is stored in the `ouptuts_0` Python variable. This variable is then passed to `Map` as the input dataset where the `stat` command is used to compute the file sizes of each input item. Next, an `AllPairs` is performed on `outputs_1` to determine the difference in file sizes for each pair-wise combination of files. Finally, `Merge` is used to tabulate `outputs_2` as the final result of executing all four `Abstractions`.

The capturing of the outputs of one abstraction and passing it to a proceeding one is how data dependencies are constructed in Weaver. Rather than having users explicitly construct the directed acyclic graph as is common in other workflow languages, the Weaver compiler extracts the structure of the workflow from these variable assignments and data dependencies automatically for the user. Therefore, to ensure a set of tasks executes before another, the end user simply has to pass the outputs of the first set of tasks as inputs or includes for the next set of tasks. This ability to connect and pipeline multiple `Abstractions` is the principle mechanism for constructing a distributed workflow in Weaver and is the fundamental design principle for the compiler's programming model.

```
# 1. MapReduce
def wordcount_mapper(key, value):
    for word in value.split():
        print('{0}\t{1}'.format(word, 1))

def wordcount_reducer(key, values):
    print('{0}\t{1}'.format(key, sum(int(v) for v in values)))

outputs_0 = MapReduce(
    mapper  = wordcount_mapper,
    reducer = wordcount_reducer,
    inputs  = Glob('weaver/*.py'),
)

# 2. Map
outputs_1 = Map('stat -c %s {IN} > {OUT}', outputs_0)

# 3. AllPairs
def diff(file_0, file_1):
    sizes = [int(open(f).read()) for f in (file_0, file_1)]
    print(sizes[0] - sizes[1])

outputs_2 = AllPairs(diff, outputs_1, outputs_1)

# 4. Merge
outputs_3 = Merge(outputs_2)
```

Figure A.4. Weaver Abstraction Examples

The first part of this listing shows how to use the `MapReduce Abstraction` *to perform a word count on the Weaver source code. The output of the* `MapReduce` *is then passed to* `Map`*, which applies the* `stat` *function to the data in order to compute the file size of each item. After this, the file sizes are then used in* `AllPairs` *to get the differences between each file. Finally, these results are tabulated using* `Merge`*.*

Each of the `Abstractions` takes advantage of the data parallelism [54] present in the patterns of computation in order maximize the amount of parallelism in the workflow. In other words, all of these `Abstractions` manifest data independence which allows the compiler to schedule the tasks associated with each pattern concurrently. Through the use of variable assignment and passing, these `Abstractions` can be pipelined to form sophisticated data intensive workflows.

A.4 Nests

The final component of the Weaver programming model is the concept of **nests**. In Weaver, all workflows consist of a workspace and a directed acyclic graph. The workspace serves as a reserved storage area for any output artifacts of the workflow, while the DAG encodes the relationships between tasks in the workflow. `Nests` are Weaver objects that represent both a workspace and DAG. Whenever an `Abstraction` is processed, it is done so in the context of a particular `Nest`, which captures any tasks produced by the abstraction being executed.

Because of this duality, the `Nest` is usually the most difficult object in the Weaver to understand for those unfamiliar with the programming model. The key to understanding this concept is to realize that a workflow consists of both a namespace and a graph. Taken together, both the namespace and the graph combine to uniquely identify a single workflow. In Weaver, each `Nest` object is associated with a particular namespace and a single DAG. Therefore, each `Nest` represents a specific workflow.

The purpose of having a `Nest` object in Weaver is to provide an execution context for the `Datasets`, `Functions`, and `Abstractions`. Whenever one of these objects is invoked it must is necessary for the compiler to know which workflow it must refer to or modify. For instance, when an `Abstraction` is executed, the compiler must determine which namespace it should use to retrieve and store any files and which DAG it should use to schedule new tasks. Because of this, all operations in Weaver are performed under the context of a single `Nest`. This active or current `Nest` can be retrieved by the `CurrentNest` function.

Implied in this discussion is the possibility of having more than one `Nest` in a workflow specification. In Weaver, users can hierarchically partition their

```
1  dataset = Glob('weaver/*.py')
2
3  with Nest('stats0'):
4      stats0 = Map('stat {IN} > {OUT}', dataset, '{BASE}.stat')
5      with Nest('stats00'):
6          stats00 = Map('stat {IN} > {OUT}', stats0, '{BASE}.stat')
7
8  with Nest('stats1'):
9      stats1 = Map('stat {IN} > {OUT}', dataset, '{BASE}.stat')
10     with Nest('stats10'):
11         stats10 = Map('stat {IN} > {OUT}', stats1, '{BASE}.stat')
12
13 Merge([stats0, stats1], '01.stats')
14 Merge([stats00, stats10], '0010.stats')
```

Figure A.5. Weaver Nests Examples

This is one of the scripts used in the Weaver test suite. In this example, we construct a hierarchical workflow where we perform `stats` *on the various files, starting with the Weaver source code and then merges all of the output. The code here demonstrates how to use* `Nest` *in conjunction with Python's* `with` *statement to manage the* `CurrentNest`*.*

workflows by creating new `Nest` objects and utilizing these new objects as the current context manager. The most straightforward way to do this is by utilizing Python's `with` statement syntax as shown in Figure A.5. In Python, the `with` statement is used as means of simplifying the use of a context manager that performs actions before and after the proceeding code block is executed. In the case of `Nest`, when used with the `with` statement the newly created object is set as the `CurrentNest` and thus any tasks that are scheduling during the execution of the code block contained by the `with` statement will be associated with the created `Nest`. When the code block exists, the previous `Nest` is restored. In this way, the user can consider this as a convenient way of managing a stack of `Nests`.

For example, on line 3 of Figure A.5, we use Python's `with` statement to construct a new `Nest` named 'stats0' and set it as the `CurrentNest`. The proceeding

`Map` operation on line 4 will use the 'stats0' `Nest` as its associated workflow and thus will schedule its tasks to that `Nest`. On line 5, we create yet another `Nest`, 'stats00', and make it active. When the `Map` on line 6 is executed, it will schedule tasks to the 'stats00' workflow. When we reach line 7, we have passed through all of the code blocks, so the `CurrentNest` is reset to the original root `Nest`. Finally, note that in lines 13 and 14 of Figure A.5, we are able to refer to variables and datasets defined `Nests` that are no longer active. Weaver handles the translation between these multiple namespaces for the end user transparently.

In summary, the Weaver programming model provides the workflow as a first-class object to the user through the use of `Nests`, which consist of both a workspace and a DAG. Users may construct `Nests` with in other `Nests` and thus partition their workflow into hierarchical structures. The utility of this data partitioning is explored later in Section 4.3.

With all of these components, the Weaver programming model provides users with a solid set of building blocks for constructing distributed data intensive scientific workflows. Using `Datasets`, users select the data they wish to process and analyze. `Functions` allow the users to specify the tools they wish to use, while `Abstractions` determine how to apply these applications to their data. Finally, `Nests` allow users to modify the structure of their workflow through hierarchical data partitioning.

APPENDIX B

WEAVER INTERNALS

Internally, Weaver's domain-specific language is implemented by overriding methods in Python's object protocol for the library components it provides. Every object in Python has a set of special methods that are defined by a strict protocol. Many of the common operators and functions in Python, such as `[]`, `+`, or `len()` correspond to specific methods in an object (`__getitem__`, `__add__`, and `__len__` respectively). For example, if a user performs `len(object)` to get the length of an object, this is basically translated internally by the interpreter to `object.__len__(self)` (Python methods include an explicit `self` argument to refer to the active object instance). To implement the DSL necessary to generate workflows, Weaver carefully overrides a few of the special object protocol methods in the `Dataset`, `Function`, and `Abstraction` components.

In the case of `Functions`, the `__call__` method is defined such that when a `Function` object is executed, it will schedule a task rather than actually perform the computation. In Python, `F(*args, *kwargs)` is basically transformed into `F.__call__(*args, *kwargs)`). By overriding the `__call__` method, we can change what happens when an object is used as a Python function. In this case, we want the Weaver `Function` to behave as a normal Python function by returning a list of results, and to act as a function in the DSL by scheduling the necessary task with the `Nest` so it will be included in the compiled DAG.

```
1  def __call__(self, inputs=None, outputs=None, arguments=None,
2      includes=None, local=False, environment=None):
3      abstraction = CurrentAbstraction()
4      nest        = CurrentNest()
5      options     = Options(environment=self.environment)
6
7      inputs      = parse_input_list(inputs)
8      outputs     = parse_output_list(outputs, inputs)
9      includes    = parse_input_list(includes) + \
10                   parse_input_list(self.includes)
11     command     = self.command_format(inputs, outputs, arguments)
12
13     if local:
14         options.local = True
15
16     if environment:
17         options.environment.update(environment)
18
19     nest.schedule(abstraction, self, command,
20         list(inputs) + list(includes), outputs, options)
21
22     return outputs
```

Figure B.1. Weaver Function `__call__` Method

In Weaver, the `Function`*'s* `__call__` *method is overridden such that is it schedules a task with the* `CurrentNest` *and returns a list of output files the task generates. Overriding this method allows Weaver* `Functions` *to look and act like normal Python functions.*

Figure B.1 shows Weaver's implementation of the `Function` `__call__` method. The top portion of the code involves setting up variables and parsing the arguments passed to the `Function`. In parsing the `outputs` argument on line 7, we are also determining what the return value of the call should be. On line 19, we schedule a task with the `CurrentNest` with the arguments parsed by the `Function`'s `__call__` method. Afterwards, we simply return the list of output files generated by this `Function`. Doing so allows us to store the results of a `Function` call and use it as an argument to another `Function` or to an `Abstraction`.

Weaver `Datasets` and `Abstractions` use a similar technique to allow these objects to behave as normal Python collections and to schedule the appropriate tasks required to execute the desired operation. In Python, any object that implements the special `__iter__` method can be used for iteration. For instance, the code `for i in object: print(i)` is translated by the Python interpreter to `for i in object.__iter__(): print(i)`. The purpose of the `__iter__` method is to return an iterator for that object. An iterator, in turn, is an object that implements the `__next__` special method and raises the `StopIteration` exception to terminate the iteration.

For `Datasets`, Weaver overrides the `__iter__` method to create *memoized futures* [5]. The implementation of Weaver's `Dataset __iter__` method is shown in Figure B.2. The core principles behind this design are: **(1)** rather than generated the dataset every time we need to iterate over it, we should cache it the first time we generate the dataset and **(2)** we should delay generating the dataset until we absolutely need it. To accomplish this, the `__iter__` method first checks to see if the cache file containing the contents of the dataset exists. If it is available and was generated after we began compiling, then the method simply returns the contents of this file in the form of a Python generator as shown on line 14. If the file does not exist, then we call the `_generate` method to retrieve a Python generator that actually constructs the `Dataset`. This means that `Datasets` in Weaver are lazily evaluated since their contents are not calculated until they are actually iterated over by another object.

To ensure that all `Datasets` memoize or cache their contents, Weaver provides a `cache_generation` Python decorator that can be applied to the `_generate` method of a `Dataset`. This decorator transparently modifies the `_generate`

```
def __iter__(self):
    # Generate the cache under any of the following conditions:
    #
    #   1. Cache file does not exist
    #   2. Cache file exists, is older than compile start time,
    #      and we are forced to do so
    debug(D_DATASET, 'Iterating on Dataset {0}'.format(self))
    if os.path.exists(self.cache_path):
        # If cache file is made after we started compiling,
        # then it is valid, so don't bother generating.
        if CurrentScript().start_time <= \
            os.stat(self.cache_path).st_ctime:
            debug(D_DATASET, 'Loading Dataset {0}'.format(self))
            return (MakeFile(f.strip(), self.nest) \
                for f in open(self.cache_path, 'r'))

        message = 'Cache file {0} already exists'.format(
                    self.cache_path)
        if CurrentScript().force:
            warn(D_DATASET, message)
        else:
            fatal(D_DATASET, message)

    debug(D_DATASET, 'Generating Dataset {0}'.format(self))
    return self._generate()
```

Figure B.2. Weaver Dataset `__iter__` Method

The overridden `Dataset __iter__` *method will first check to see if cached listing of the* `Dataset` *exists. If it does and it was generated after we started compiling, then we return this cached listing, otherwise we report an error. If the cache does not exist, then we generate it by calling the* `_generate` *method.*

method such that any items yielded by the original `_generate` method is captured and stored in a cache file. Figure B.3 displays the implementation of the `Map Abstraction's _generate` method. In this implementation, `Map` simply iterates over the inputs and outputs and calls the `Function` it was given with these arguments to schedule the appropriate tasks. Because the `cache_generation` decorator is used, the items *yielded* by `Map's _generate` method are cached to a file for retrieval by the `__iter__` method in subsequent iterations.

```
@cache_generation
def _generate(self):
    with self:
        debug(D_ABSTRACTION, 'Generating Abstraction {0}'.format(
            self))

        function = parse_function(self.function)
        inputs   = parse_input_list(self.inputs)
        outputs  = parse_output_list(self.outputs, inputs)
        includes = parse_input_list(self.includes)

        for i, o in zip(inputs, outputs):
            with Options(local=self.options.local,
                         collect=[i] if self.collect else None):
                yield function(i, o, None, includes)
```

Figure B.3. Weaver Map `_generate` Method

All `Abstractions` *in Weaver are also* `Datasets`*, therefore the implementation of the actual* `Abstraction` *is found in the* `_generate` *method. This example shows the implementation of the* `Map` *abstraction. Note the use of the* `cache_generation` *Python decorator which will store the results of the method in a cache file for later retrieval by the* `__iter__` *method.*

In addition to overriding Python object protocol methods, the Weaver compiler depends heavily on Python's `with` statement to set and restore various global variables that constitute the current workflow context. For instance, it maintains stacks for the active `Nest`, `Abstraction`, `Script`, and `Options` objects (a `Script` object refers to the compiler script and is used to access compiler flags, while the `Options` object stores various task parameters and is discussed in Chapter 7). To access the current instance of these objects, the compiler provides the `CurrentNest`, `CurrentAbstraction`, `CurrentScript`, and `CurrentOptions` functions respectively. Using the `with` statement along with these stacks allows the compiler to track and access these objects in a sane manner, without having to pass all of these parameters everywhere.

Evaluating a script and generating the graph as a side-effect of that evaluation is a technique similar that used by GRID superscalar [104] except that Weaver is an optimizing compiler, rather than a run-time system. Moreover, the Weaver compiler utilizes lazy evaluation and caching to implement the core elements of the domain-specific language. By overriding some of Python's special methods in the `Dataset`, `Function`, and `Abstraction` classes, Weaver is able to have these objects behave as normal Python iterators and functions and at the same time generate tasks for the workflow.

BIBLIOGRAPHY

1. The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman, 2002. URL `http://www.cs.wisc.edu/condor/dagman`.

2. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

3. M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A portable abstraction for cluster, cloud, and grid computing. Technical Report TR-2011-02, Department of Computer Science and Engineering, University of Notre Dame, 2011.

4. A. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

5. H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM. doi: 10.1145/800228.806932. URL `http://doi.acm.org/10.1145/800228.806932`.

6. H. G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *SIGPLAN Not.*, 30:17–20, September 1995. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/214448.214454. URL `http://doi.acm.org/10.1145/214448.214454`.

7. H. Baker Jr and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12(8):55–59, 1977.

8. R. Barga and L. Digiampietri. Automatic generation of workflow provenance. *Provenance and Annotation of Data*, pages 1–9, 2006.

9. A. Barker and J. van Hemert. Scientific Workflow: A Survey and Research Directions. In R. Wyrzykowski and et al., editors, *Seventh International Conference on Parallel Processing and Applied Mathematics, Revised Selected Papers*, volume 4967 of *LNCS*, pages 746–753. Springer, 2008.

10. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 329–338, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54023. URL `http://doi.acm.org/10.1145/53990.54023`.

11. D. Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, pages 176–187. Springer, 1987.

12. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM. ISBN 0-89791-902-5.

13. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN Notices*, volume 30, August 1995.

14. H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

15. H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988. ISSN 1097-024X. doi: 10.1002/spe.4380180902.

16. F. P. Brooks, Jr. The computer scientist as toolsmith ii. *Commun. ACM*, 39(3):61–68, Mar. 1996. ISSN 0001-0782. doi: 10.1145/227234.227243. URL `http://doi.acm.org/10.1145/227234.227243`.

17. H. Bui, M. Kelly, C. Lyon, M. Pasquier, D. Thomas, P. Flynn, and D. Thain. Experience with BXGrid: A Data Repository and Computing Grid for Biometrics Research. *Journal of Cluster Computing*, 12(4):373, 2009.

18. H. Bui, P. Bui, P. Flynn, and D. Thain. ROARS: A Scalable Repository for Data Intensive Scientific Computing. In *The Third International Workshop on Data Intensive Distributed Computing at ACM HPDC 2010*, 2010.

19. H. Bui, P. Bui, P. Flynn, and D. Thain. ROARS: A scalable repository for data intensive scientific computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 766–775, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8.

20. H. Bui, D. Wright, C. Helm, R. Witty, P. Flynn, and D. Thain. Towards long term data quality in a large scale biometrics experiment. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed*

Computing, HPDC '10, pages 565–572, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8.

21. P. Bui, L. Yu, and D. Thain. Weaver: Integrating distributed computing abstractions into scientific workflows using Python. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 636–643, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8.

22. P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.

23. P. Bui, L. Yu, A. Thrasher, R. Carmichael, I. Lanc, P. Donnelly, and D. Thain. Scripting distributed scientific workflows using Weaver. *Concurrency and Computation: Practice and Experience*, 2011.

24. D. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

25. Cascading. http://www.cascading.org/, 2010. URL `http://www.cascading.org/`.

26. R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008. ISSN 2150-8097.

27. B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

28. D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM. doi: 10.1145/800296.811515. URL `http://doi.acm.org/10.1145/800296.811515`.

29. C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3.

30. P. Chang and W. Hwu. Inline function expansion for compiling c programs. *ACM SIGPLAN Notices*, 24(7):246–257, 1989.

31. D. Chase. Safety consideration for storage allocation optimizations. *ACM SIGPLAN Notices*, 23(7):1–10, 1988.

32. D. Cieslak, N. Chawla, and D. Thain. Troubleshooting Thousands of Jobs on Production Grids Using Data Mining Techniques. In *IEEE Grid Computing*, pages 217–224, 2008.

33. C. Collberg, J. Hartman, S. Babu, and S. Udupa. Slinky: Static linking reloaded. In *USENIX 2005 Annual Technical Conference*, pages 309–322, 2005.

34. L. Dagum and R. Menon. OpenMP: An industry standard api for shared memory programming. *IEEE Computational Science and Engineering*, 1998.

35. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, 2004.

36. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3), 2005.

37. G. Dong, R. Hull, B. Kumar, J. Su, and G. Zhou. A framework for optimizing distributed workflow executions. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin / Heidelberg, 2000.

38. J. Dongarra and A. Hinds. Unrolling loops in fortran. *Software: Practice and Experience*, 9(3):219–226, 1979.

39. J. J. Dongarra and D. W. Walker. MPI: A standard message passing interface. *Supercomputer*, pages 56–68, January 1996.

40. N. Dun, K. Taura, and A. Yonezawa. Paratrac: a fine-grained profiler for data-intensive workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 37–48, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851482. URL `http://doi.acm.org/10.1145/1851476.1851482`.

41. J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3535-7. doi: 10.1109/eScience.2008.59. URL `http://dx.doi.org/10.1109/eScience.2008.59`.

42. W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (bpel). *Journal of Grid Computing*, 3:283–304, 2005.

43. S. Feldman. Make – A Program for Maintaining Computer Programs. *Software: Practice and Experience*, 9:255–265, November 1978.

44. S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.

45. C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27:68–76, April 1992. ISSN 0362-1340.

46. A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

47. S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 18–27, New York, NY, USA, 1999. ACM. ISBN 1-58113-111-9.

48. W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 35–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8. URL `http://dl.acm.org/citation.cfm?id=560889.792378`.

49. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. ISBN 0321246780.

50. S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM. ISBN 0-89791-074-5. doi: 10.1145/800230.806987. URL `http://doi.acm.org/10.1145/800230.806987`.

51. Hadoop. http://hadoop.apache.org/, 2007. URL `http://hadoop.apache.org/`.

52. R. Hank, W.-m. Hwu, and B. Rau. Region-based compilation: Introduction, motivation, and initial experience. *International Journal of Parallel Programming*, 25:113–146, 1997. ISSN 0885-7458.

53. T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009. URL `http://research.microsoft.com/en-us/collaboration/fourthparadigm/`.

54. W. Hillis and G. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

55. W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Softw. Pract. Exper.*, 21(4):375–390, Apr. 1991. ISSN 0038-0644. doi: 10.1002/spe.4380210404. URL `http://dx.doi.org/10.1002/spe.4380210404`.

56. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys.*, 6(1):51–81, February 1988.

57. D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 187–200. ACM, 1990.

58. R. Ierusalimschy, L. De Figueiredo, and W. Filho. Lua-an extensible extension language. *Software Practice and Experience*, 26(6):635–652, 1996.

59. M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2.

60. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.

61. James da Silva. http://netbsd.gw.com/cgi-bin/man-cgi?crunchgen++NetBSD-current, 1994. URL `http://netbsd.gw.com/cgi-bin/man-cgi?crunchgen++NetBSD-current`.

62. W. Keller. Mapping objects to tables. In *Proceedings of Second European Conference on Pattern Languages of Programming (EuroPLoP'97). Siemens Technical Report*, volume 120. Citeseer, 1997.

63. Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31:406–471, December 1999. ISSN 0360-0300.

64. R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

65. M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *ACM Sigplan Notices*, volume 23, pages 318–328. ACM, 1988.

66. M. Lammie, D. Thain, and P. Brenner. Scheduling Grid Workloads on Multicore Clusters to Minimize Energy and Maximize Performance. In *IEEE Grid Computing*, 2009.

67. I. Lanc, P. Bui, D. Thain, and S. Emrich. Adapting Bioinformatics Applications for Heterogeneous Systems: A Case Study. In *Emerging Computational Methods for the Life Sciences Workshop at ACM HPDC*, pages 7–13, 2011.

68. J. Larus. Whole program paths. *ACM SIGPLAN Notices*, 34(5):259–269, 1999.

69. C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL `http://dl.acm.org/citation.cfm?id=977395.977673`.

70. J. Levon and P. Elie. Oprofile: A system profiler for linux, 2004.

71. C. C. Lian, F. Tang, P. Issac, and A. Krishnan. Gel: Grid execution language. *Journal of Parallel and Distributed Computing*, 65:2005, 2005.

72. J. Linderoth, S. Kulkarni, J.-P. Goux, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *IEEE High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.

73. T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

74. P. Louridas. Orchestrating web services with bpel. *IEEE Software*, 25:85–87, 2008. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2008.42.

75. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10): 1039–1065, 2006.

76. E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0.

77. P. Miller. Recursive make considered harmful, 1997.

78. C. Moretti, J. Bulosan, D. Thain, and P. Flynn. All-Pairs: An Abstraction for Data Intensive Cloud Computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2008.

79. C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain. All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids. *IEEE Transactions on Parallel and Distributed Systems*, 21(1):33–46, 2010.

80. R. Morrison, M. Atkinson, A. Brown, and A. Dearle. On the classification of binding mechanisms. *Information Processing Letters*, 34(1):51–55, 1990.

81. A. Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, pages 11–15, 2009.

82. D. Murray and S. Hand. Scripting the cloud with skywriting. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 12–12. USENIX Association, 2010.

83. D. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of NSDI*, 2011.

84. A. MySQL. Mysql, 2005.

85. T. Oinn, M. Addis, J. Ferris, D. Marvin, T. Carver, M. R. Pocock, and A. Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:2004, 2004.

86. T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18:1067–1100, August 2006. ISSN 1532-0626. doi: 10.1002/cpe.v18:10. URL `http://portal.acm.org/citation.cfm?id=1148437.1148448`.

87. T. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing, 2006.

88. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6.

89. Oozie. http://yahoo.github.com/oozie/, 2010. URL `http://yahoo.github.com/oozie/`.

90. M. Owens. *The definitive guide to SQLite*. Apress, 2006.

91. S. Papadimitriou and J. Sun. Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 512–521, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3502-9. doi: 10.1109/ICDM.2008.142. URL `http://dx.doi.org/10.1109/ICDM.2008.142`.

92. K. Pettis and R. Hansen. Profile guided code positioning. In *ACM SIGPLAN Notices*, volume 25, pages 16–27. ACM, 1990.

93. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 13(4):227–298.

94. C. D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 252–263, New York, NY, USA, 1991. ACM. ISBN 0-89791-434-1.

95. R. Prodan and T. Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 687–694. ACM, 2005.

96. Python Programming Language. http://www.python.org/, 2010. URL `http://www.python.org/`.

97. I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 43:1–43:12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362680. URL `http://doi.acm.org/10.1145/1362622.1362680`.

98. C. Rasmussen, M. Sottile, S. Shende, and A. Malony. Bridging the language gap in scientific computing: The chasm approach. *Concurrency and Computation: Practice and Experience*, 18(2):151–162, 2006.

99. M. Samek. Portable inheritance and polymorphism in c. *Embedded Systems Programming*, 10:54–67, 1997.

100. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *USENIX Summer Technical Conference*, pages 119–130, 1985.

101. C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with vistrails. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1251–1254, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6.

102. H. J. Siegel. A model of simd machines and a comparison of various interconnection networks. *IEEE Trans. Comput.*, 28(12):907–917, Dec. 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675280. URL `http://dx.doi.org/10.1109/TC.1979.1675280`.

103. G. Singh, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *Journal of Grid Computing*, 3:201–219, 2005. ISSN 1570-7873.

104. R. Sirvent, J. M. Pérez, R. M. Badia, and J. Labarta. Automatic grid workflow based on imperative programming languages: Research articles. *Concurr. Comput. : Pract. Exper.*, 18:1169–1186, August 2006. ISSN 1532-0626. doi: 10.1002/cpe.v18:10.

105. SQLAlchemy. http://sqlalchemy.org/, 2010. URL `http://sqlalchemy.org/`.

106. R. Stallman. Using and porting the gnu compiler collection. *Free Software Foundation*, 59:02111–1307, 1989.

107. R. Stallman and R. Pesch. *Using GDB: A guide to the GNU source-level debugger*. Free software foundation, 1991.

108. G. Steele Jr. Parallel programming and parallel abstractions in fortress. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, page 157. IEEE, 2005.

109. G. Steele Jr and G. Sussman. Lambda: The ultimate imperative. Technical report, DTIC Document, 1976.

110. R. Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(Suppl 12):S1, 2010. ISSN 1471-2105. doi: 10.1186/1471-2105-11-S12-S1. URL `http://www.biomedcentral.com/1471-2105/11/S12/S1`.

111. D. Thain and M. Livny. How to Measure a Large Open Source Distributed System. *Concurrency and Computation: Practice and Experience*, 18(15): 1989–2019, 2006.

112. D. Thain and M. Livny. Parrot: An Application Environment for Data-Intensive Computing. *Scalable Computing: Practice and Experience*, 6(3): 9–18, 2005.

113. D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In gri [2].

114. A. Thrasher, R. Carmichael, P. Bui, L. Yu, D. Thain, and S. Emrich. Taming complex bioinformatics workflows with Weaver, Makeflow, and Starch. In *Workflows in Support of Large-Scale Science (WORKS), 2010 5th Workshop on*, pages 1 –6, 2010.

115. Valery Reznic. http://statifier.sourceforge.net/, 2010. URL `http://statifier.sourceforge.net/`.

116. G. von Laszewski and M. Hategan. Workflow concepts of the java cog kit. *Journal of Grid Computing*, 3:239–258, 2005. ISSN 1570-7873.

117. J. Wang, D. Crawl, and I. Altintas. Kepler + hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 12:1–12:8, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-717-2. doi: 10.1145/1645164.1645176. URL `http://doi.acm.org/10.1145/1645164.1645176`.

118. B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel. Sedna: A bpel-based environment for visual scientific workflow modelling. In *In Workflows for eScience - Scientific Workflows for Grids*. Springer Verlag, 2007.

119. M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42:50–60, 2009. ISSN 0018-9162.

120. M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011. URL `http://linkinghub.elsevier.com/retrieve/pii/S0167819111000524`.

121. P. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 1992. URL http://dx.doi.org/10.1007/BFb0017182.

122. H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247602. URL http://doi.acm.org/10.1145/1247480.1247602.

123. J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3:171–200, 2006.

124. L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain. Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions. *Journal of Cluster Computing*, 13(3):243–256, 2010.

125. C. Zhang and H. De Sterck. Cloudwf: A computational workflow system for clouds based on hadoop. In M. Jaatun, G. Zhao, and C. Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 393–404. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-10664-4.

126. Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data. In *SIGMOD*, 2005.