Tracing Lineage Beyond Relational Operators *

Mingwu Zhang† Xiangyu Zhang† Xiang Zhang‡ Sunil Prabhakar†

†Department of Computer Sciences
Purdue University
West Lafayette, Indiana, USA
{mzhang2, xyzhang, sunil}@cs.purdue.edu

‡Bindley Bioscience Center
Purdue University
West Lafayette, Indiana, USA
{zhang100}@purdue.edu

ABSTRACT

Tracing the lineage of data is an important requirement for establishing the quality and validity of data. Recently, the problem of data provenance has been increasingly addressed in database research. Earlier work has been limited to the lineage of data as it is manipulated using relational operations within an RDBMS. While this captures a very important aspect of scientific data processing, the existing work is incapable of handling the equally important, and prevalent, cases where the data is processed by non-relational operations. This is particularly common in scientific data where sophisticated processing is achieved by programs that are not part of a DBMS. The problem of tracking lineage when non-relational operators are used to process the data is particularly challenging since there is potentially no constraint on the nature of the processing. In this paper we propose a novel technique that overcomes this significant barrier and enables the tracing of lineage of data generated by an arbitrary function. Our technique works directly with the executable code of the function and does not require any high-level description of the function or even the source code. We establish the feasibility of our approach on a typical application and demonstrate that the technique is able to discern the correct lineage. Furthermore, it is shown that the method can help identify limitations in the function itself.

1. INTRODUCTION

With the advance of high-throughput experimental technology, scientists are tackling large scale experiments and producing enormous amounts of data. Web technology allows scientists to collaborate and share data – further increasing the amount of available data. To increase the usability of this data, it is essential to know the provenance of the data – how it was generated, using what tools, what parameters were used, etc. This information is often termed *Provenance* or *Lineage* of the data. Lineage information can be used to estimate the quality, reliability, and applicability of data to a given task. An important aspect of data provenance is *Relationship* [23], which has been defined as "Information on how one data item in a process relates to another." Despite the importance of these relationships between input and output data, acquiring them remains a challenge which has not been addressed by the existing work [17, 23].

Lineage can be categorized into *coarse-grained lineage* and *fine-grained lineage*. Coarse-grained lineage records the procedures used to transform the data, the parameters used and a general description of the input and output data. Coarse grained lineage is also referred to as work-flow in literature. To improve scientific collaboration, Workflow Management System and Grid computation are used to simplify access to computational resources and experimental results over distributed systems[15, 14, 24, 11]. Many prototype systems such as Chimera[11], ^{My}Grid[24], and Geo-Opera [8] have been developed. There is a subtle difference between workflow and lineage. Workflow defines a plan for desired processing before it actually happens. Lineage, on the other hand, describes the relationship between data products and data transformations after processing has occurred. Coarse-grained lineage is useful in many applications. However, applications such as the scientific computations in [25, 21] require fine-grained lineage. Coarse-grained lineage is insufficient since detailed information of how individual output elements are derived from a certain subset of input elements is desired.

Lineage tracing in the context of database systems has been extensively studied [12, 10, 13]. These algorithms can trace fine-grained lineage only when the data is produced by relational operators within a DBMS. Consequently, they cannot be applied to tracing data lineage when non-relational operators are employed as is often the case with scientific applications. For example, a workflow may involve programs maintained and distributed at different research groups, but shared within the grid. The program could be an executable or a web service implementing an arbitrary function that the user knows little about. Even though the data may be stored in a database, the program used to derive the data usually resides outside the database, or at best as a stored procedure. To the best of our knowledge, there is currently no technique that enables lineage tracing for these "black box" functions.

A similar challenge is also seen in data mining and data

*This work is supported by NSF grant number 0534702, 0242421 and AFOSR award number FA9550-06-1-0099

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

cleansing applications. For many applications, data cleaning is an important first step. It has been reported that the data cleaning procedure takes more than half the total time in the construction of a data warehouse. Extensive research has been conducted on how to resolve inconsistencies and missing values in the data. However, after the data is cleaned and stored in the database, the information of how the data is cleaned is not stored in the database and is lost forever. For example, the missing value could be replaced by a most likely value or derived from a model, but once the data has made it to the database, this data is treated as if it is the real data. In many cases, the data used to replace the missing value may be incorrect. It is important to maintain this information if one has doubts about the data. Since the data cleaning procedures are usually performed by a program without relational semantics, it is currently difficult to obtain this information.

Despite the importance of the problem, there has been very limited work that has addressed the problem of tracing lineage when arbitrary functions are used – this is largely due to the difficulty of the problem. In [25], Wooddruff and Stonebraker use reverse functions to compute the mappings from output to input. A reverse function returns the set of input data that is used to generate a given output data item. When a reverse function is not available, a weak reverse function is used to compute a set of input that is a superset or subset of the data used to compute the output. A verification function is also used to refine the set. Marathe [21] apply rewrite rules to AML (Array Manipulation Language) expression in order to trace fine-grained lineage for array data. This lineage, however, may contain false positives. These solutions have been shown to be effective in certain scenarios. However, they have their inherent limitations. First, reversibility is not a universal characteristic of data processing queries/functions. Even when a weak reverse function can be found, it will not be very useful if the exact data items can not be identified. Second, in order to design reverse queries/functions, a comprehensive understanding of the data processing procedures is a pre-requisite, which makes the solutions application-specific and hard to automate. The situation becomes worse when it comes to legacy code because they are often harder to understand. Third, coding the reverse queries/functions entails non-trivial efforts, which thwart the application of these techniques.

To the best of our knowledge, there is no existing work that is able to automatically infer the connections between input and output for arbitrary functions. In this paper, we propose the first such technique. The key idea of our technique is the observation that the program binary that implements a function is a valuable source of information that connects input and output. Therefore, tracing program executions reveals how output is derived from input.

While this is a very promising direction, the design and implementation is non-trivial. In this paper, we take advantage of recent advances in dynamic program analysis and propose fine grained lineage tracing through dynamic program slicing. Dynamic program slicing is a technique originally designed to debug program errors. Given an executable, dynamic program slicing is able to trace the set of statements that have been involved in computing a specific value at a particular execution point, thus helping the programmer to find the bug. Using dynamic program slicing to trace fine-grained lineage has many immediate advantages: it is a completely automated general solution and does not require any user input or domain expertise on the data processing functions; it can simply work on compiled binaries without access to the source code, and the traced fine grained lineage is accurate.

The only barrier to realizing this intuitive idea is the cost. Fortunately, recent progress in program tracing, especially in dynamic program slicing, enables tracing fine grained lineage with reasonable cost. It is worth pointing out that part of the overhead of our system stems from the underlying dynamic program analysis engine which is based on a single-core machine and not highly optimized. As a result, it is usually several times slower than an industry-strength engine. Even in the absence of support from a more efficient engine, the contribution of this paper is significant since it provides a new functionality that is currently not available. For most applications that require lineage information – the availability of the information is more crucial than the runtime cost of computing it. At the same time, it is a simple matter to support rapid query processing without lineage tracing while at the same time having a separate, slower computation that generates the lineage information in the background. In this fashion, query results are available immediately while the lineage information is generated a little later. We show in this paper, that even though lineage tracing is slower than query processing, it remains at acceptable levels for all the applications that we have considered – we are certain that this is a price that these applications are willing to pay for obtaining valuable lineage information that has not been available earlier. This is strongly supported by our experiments. Overall, this paper makes the following contributions:

- We develop the first fine-grained lineage tracing algorithm that can be applied to any arbitrary function without human input or access to source code. We describe how the ideas of dynamic slicing for debugging programs are adapted to provide fine-grained lineage.
- We implement the system and apply it to real applications. Our experiments show that the overhead is acceptable. Our case study demonstrates that the traced fine grained lineage information is highly effective and greatly benefits a biologist in analyzing and understanding the results.

2. MOTIVATION

In this section we describe a motivating data processing application of non-relational data processing for which lineage tracing is both important and challenging. Liquid Chromatography Mass Spectrometry (LC-MS) is an effective technique used in protein biomarker discovery in cancer research [30, 29]. Figure 1 shows the various steps involved in LC-MS. A biomarker is a protein which undergoes changes in structure or concentration in diseased samples. Identified biomarkers can be used in diagnoses and drug design. To detect these biomarkers, proteins from cancer patients and normal people are digested into smaller pieces called peptides. These two samples of peptides are "labeled" by attaching chemicals to them. The normal and diseased groups are distinguished by having different isotopes (same chemical structure but different molecular mass) in their labels. The labeled cancer and normal samples are then

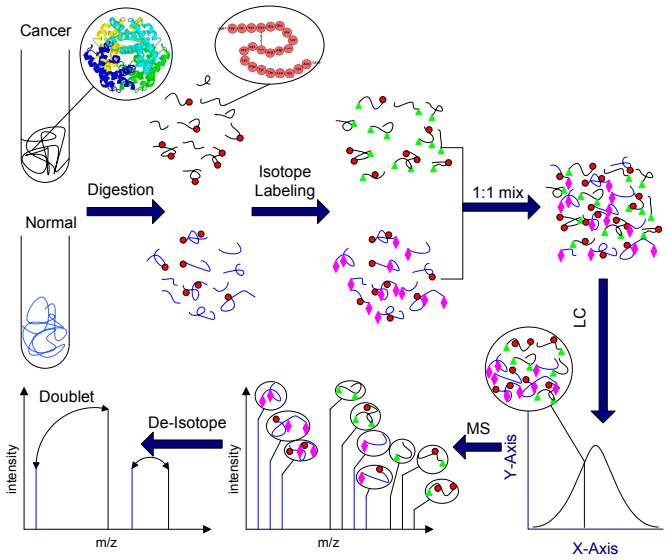


Figure 1: An overview of the LC-MS process.

mixed in equal parts. An ionization process is used to attach charges to the peptides. Due to the nature of this process, different molecules of the same peptide can end up with different charges, thereby producing different m/z ratios. This mixture is then subjected to the LC-MS process. This process is only able to measure the ratio of molecular mass to the charge of the peptide.

In an ideal situation, each peptide would produce two peaks – one corresponding to the normal peptide marked with a light label and a second corresponding to the cancer sample marked with a heavy label. These peaks would differ in mass by the difference in the label weights and are called a *doublet*. Unfortunately, the data from the spectrometer is not so clean. There are two main reasons (in addition to the difficulty of wet-bench experimentation): charges and naturally occurring isotopes. The charge that gets attached to a given peptide is not fixed. Thus a peptide with mass m_0 may show up at a m/z ratio of m_0, $m_0/2$, $m_0/3$, etc. depending upon its charge in the experiment.

Furthermore, due to naturally occurring isotopes, different molecules of the same peptide may have different molecular mass and thereby results in a cluster of peaks, called isotopic peaks. [1] The mass difference between two adjacent isotopic peaks equals to 1Dalton. Assuming that isotopes take on the same charge, with a charge of +1, the m/z difference between these peaks will be 1, with a charge of +2, the difference will be 0.5, and with a charge of +3, the difference will be 0.33 etc. Thus, we see that *a single peptide can contribute to multiple peaks in the LC-MS output. Also, a single observed peak could contain contributions from multiple peptides (due to multiple isotopes and charges).*

[1]For example, Carbon atoms usually have an atomic weight of 12. However, there is usually a small fraction of Carbon atoms with weight 13 (and even fewer with weight 14). Depending upon the number of C^{13} atoms in a given molecule, the overall molecular weight of that molecule may differ. Similarly Nitrogen, which is commonly found as N^{14}, also has a less frequent isotope: N^{15}. Thus depending upon the number and type of isotopes that make up a given molecule, we get multiple molecular weights for the same peptide. Note however, that the typical ratio of the occurrence of these isotopes is usually known.

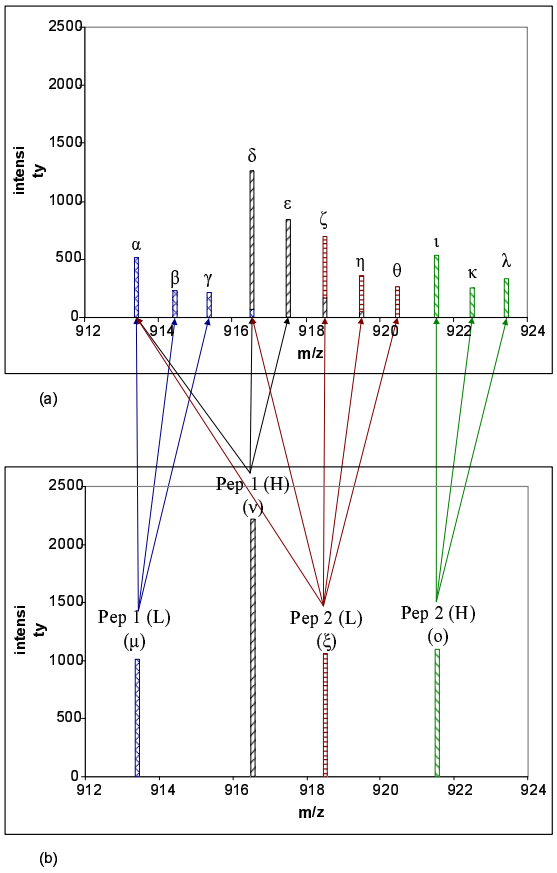


Figure 2: Sample Mass Spectrometry results (a) raw data and (b) analysis results.

The spectrometer produces an output similar to that shown in Figure 2(a). The x-axis of the graph shows the ratio of the molecular mass to the charge (m/z ratio) of the various peptides that were detected by the spectrometer. Molecular mass is measured in Dalton (Da) and charges are integer values (typically +1, +2, +3 or +4). The y-axis shows the intensity (concentration) of that particular m/z value. Each peak is labeled with a greek symbol to ease exposition. For example, the left-most peak corresponds to a m/z ratio of 913.437, we will refer to this as Peak α.

De-isotoping functions are employed to process the raw spectrometer output in order to identify the peptides that could have generated the observed pattern of peaks. In this output, the same type of peptides generated from normal and disease samples will appear in the same mass spectrum as a doublet. The intensity ratio of the doublet indicates the relative concentration of proteins from which the peptides were generated. If the ratio is not equal to the expected ratio (the ratio in which the samples were mixed), then the protein which generated the peptide may be a potential biomarker.

Due to the complexity of the process and the many factors that can lead to errors, the de-isotoping functions are heuristics that scientists have developed over a period of time. Not surprisingly, LC-MS has been known to produce a large percentage of false positives. Many factors contribute to the problem of many false positives: The data quality may be

poor; The heuristics used in the algorithm may not handle some situations; The design of the algorithm may contain flaws; and there could be human errors that are not easy to detect. It is evident that the quantification of peak intensity is critical for the success of the experiment. Eliminating false positives is important since the results of the LC-MS will determine in what direction the subsequent research will proceed – typically involving significant effort and expense. It is important for scientists to have high confidence that a potential biomarker is worthy of further analysis. The availability of fine-grained lineage can significantly improve scientists' ability to eliminate false positives.

Algorithm 1 shows the pseudo-code for the state-of-the-art algorithm for the de-isotope procedure. For each peak P in the spectrum, up to six isotopic peaks are identified. The intensity of each isotopic peak is compared against a theoretical threshold that is computed from *P.intensity* and a constant H, which is indexed by i and *P.intensity*. If it equals the threshold, this intensity is aggregated to *P.intensity* and the peak is removed from the spectrum. Otherwise, the threshold intensity is added to *P.intensity* and subtracted from the isotopic peak's intensity.

The de-isotope procedure cleans up the raw LC-MS output and generates a spectrum that has no isotopic peaks. Thus a peak in the output corresponds to the sum of the intensities of all isotopic peaks for the same peptide. It should be noted that this procedure is based upon domain expertise and heuristics and may itself have some errors. A sample result is shown in Figure 2(b). For example, the intensity of peak μ denoted as P^{μ}, is computed by the following equation:

$$P^{\mu} = (1 + c_2) \cdot P^{\alpha} + P^{\beta} + P^{\gamma}$$

Similarly, the intensity of peak ν is computed as:

$$P^{\nu} = (1 + c_1' + c_2') \cdot (P^{\delta} - c_2 \cdot P^{\alpha}) + P^{\varepsilon}$$

The values of the constants used, and the actual peaks that contribute in each case, depend upon the processing details that are buried in the complex procedure. This significantly complicates the ability to automatically infer the relationships between input and output. It is obvious that no reverse function exists for the functions listed above. One possible weak function is to compute all six possible isotopic peaks, which will include $P^{\alpha}, P^{\beta}, P^{\gamma}, P^{\delta}, P^{\varepsilon}, P^{\zeta}, P^{\eta}$. This is a superset of the real lineage. The other possible weak reverse function is to find the peak with the same m/z, which is P^{α}. This set is a subset of the real lineage. Neither reverse weak function gives a satisfactory result. In addition, there is no good verification function to refine the result produced by weak reverse function. In this case, the true function used to compute the intensity will depend on many conditions and would have to be dynamically generated. Note that coarser grained lineage such as lineage at method level does not help either because it is the different execution paths within a single method that result in various dynamic functions.

Algorithm 1 De-isotope

```
1: for each peak P in the spectrum do
2:   Ch=F(P) /*Compute the charge of P*/
3:   M[]=G(Ch,P) /* Find the next up to 6 isotopic peaks*/
4:   for each M[i] do
5:     T = H(P,i) × P.intensity /*H(...) is the constant ratio for calculating theoretical isotopic peak intensity*/
6:     if (M[i].intensity ≡ T) then
7:       P.intensity+ = M[i].intensity
8:       remove peak M[i] from the spectrum
9:     else
10:      P.intensity+ = T
11:      M[i].intensity = M[i].intensity − T
12:    end if
13:  end for
14:  print(..., P.intensity, ...)
15:  remove P from spectrum
16: end for
```

3. AUTOMATED LINEAGE TRACING

In this section we present our new approach for automatic tracing of fine-grained lineage through run-time analysis. This approach is motivated by the technique of dynamic slicing that is used as a debugging tool [20]. Dynamic slicing is able to identify a subset of program statements that are involved in producing erroneous executions. The goal of lineage tracing is rather different in that we are interested in identifying the connections between input and output data for a program. Although not straight-forward, we show that it is possible to adapt the technique of dynamic slicing for our purpose. Before we discuss how this is achieved, we present a very brief description of dynamic slicing as used for debugging. Interested readers are referred to [20] for further details.

3.1 Dynamic Slicing

Dynamic slicing operates by observing the execution of the program on given input data. The goal is to be able to determine which statements are responsible for the execution having reached a given statement. Each statement is identified by a line number, s. Since a given statement may be executed many times in a given execution, each execution of Statement s is identified with a numeric subscript: s_i for the ith execution.

DEFINITION 1. *Given a program execution, the* **dynamic slice** *of an execution point of s_i, which denotes the i^{th} execution instance of statement s, is the set of statements that directly or indirectly affected the execution of s_i.*

In order to identify the set of relevant statements, dynamic slicing captures the exercised dependencies between statement executions. The dependencies can be categorized into two types, *data dependence* and *control dependence*.

DEFINITION 2. *A statement execution instance s_i* **data depends** *on another statement execution instance t_j if and only if a variable is defined at t_j and then used at s_i.*

In the execution presented in Figure 3, for example, there is a data dependence from 6_0 to 5_0 since T is defined at 5_0 and then used at 6_0. Note that a variable is *defined* if it appears in the left hand side of an assignment statement.

Besides data dependence, another type of dependence captured by dynamic slicing is called control dependence.

DEFINITION 3. *A statement execution instance s_i* **control depends** *on t_j if and only if (1) statement t is a predicate statement and (2) the execution of s_i is the result of the branch outcome of t_j.*

```
...
4_0.  for M[0];
5_0.     T = ... P.intensity
6_0.     if (T ≡ M[0].intensity)
7_0.        P.intensity+ = M[0].intensity
8_0.        ...
4_1.  for M[1];
5_1.     T = ... P.intensity
6_1.     if (T ≡ M[1].intensity)
9_0.     else
10_0.       P.intensity+ = T
11_0.       ...
4_2.  for NULL;
14_0. print (..., P.intensity, ...)
...
```

Figure 3: Execution Trace of Algorithm 1

For example in Figure 3, 7_0 and 8_0 control depend on 6_0. More details on how to identify control dependence at runtime can be found in [28].

The dynamic slice of an executed statement s_i consists of s_i and the dynamic slices of all the executed statements that s_i data or control depends on. Therefore, the dynamic slice of 14_0 contains 14_0, 10_0, 6_1, 5_1, 4_1, 7_0, 6_0, 5_0 and 4_0.

3.2 Tracing Data Lineage

For the case of lineage tracing we are interested in determining the set of *input items* that are involved in computing a certain value at a particular execution point. In this section, we adapt the dynamic slicing technique for data lineage computation.

We start by defining *data lineage* in terms of program execution.

DEFINITION 4. *Given a program execution, the* **data lineage** *of v at an execution point of s_i, denoted as $DL(v@s_i)$, is the set of input items that are directly or indirectly involved in computation of the value of v at s_i through data or control dependences.*

We also use $DL(s_i)$ to denote the data lineage of the left hand side expression of s_i. For example,

$$DL(P@14_0) = \{P, M[0], M[1]\}$$

Dynamic slices are usually computed by first constructing a dynamic program dependence graph [6], in which an edge reveals a data/control dependence between two statement instances, and then traversing the graph to identify the set of reachable statement instances. This method suffers from the unbounded size of the constructed dependence graph. More recently, it has been shown that dynamic slices can be computed in a forward manner [9, 27], in which slices are continuously propagated and updated as the execution proceeds. While this method mitigates the space problem, dynamic slices are often so large that expensive operations have to be performed at each step of the execution in order to update the slices.

Fortunately, in lineage tracing, *it is not necessary to trace statement executions.* Consider the example below. It is obvious the lineage set of $OUTPUT$ contains only $INPUT[0]$. However, all statement executions should be contained in the dynamic slice of $OUTPUT$ because they directly/indirectly contributed to the value of $OUTPUT$.

```
1_0: x = INPUT[0];
2_0: x = x + 1;
3_0: OUTPUT = x;
```

In other words, if well designed, lineage tracing can be much more efficient than dynamic slicing.

Next we describe how data lineage is computed during program execution. The basic idea is that *the set of input elements that is relevant to the right hand side variable at s_i is the union of the relevant input sets of all the statement instances which s_i data or control depends on.* In other words, all the input items that are relevant to some operand of s_i or the predicate that controls the execution of s_i are considered as relevant to s_i as well.

For simplicity of explanation, let

$$s_i : \texttt{dest} =? \ t_j \ : f(\texttt{use}_0, \texttt{use}_1, ..., \texttt{use}_n)$$

be the executed statement instance s_i, which assigns a value to variable `dest` by using the variables of $\texttt{use}_0$,$\texttt{use}_1$, ..., and $\texttt{use}_n$, and s_i control depends on t_j. For example, the statement instance 10_0 can be denoted as

$$10_0 : \texttt{P.intensity} =? \ 6_1 \ : f(\texttt{P.intensity}, T)$$

because it control depends on 6_1 and defines `P.intensity` using T and the old `P.intensity`.

Let $DEF(x)$ be the latest statement instance that defines x. The computation of data lineage can be represented by the following equations:

$$\begin{aligned} DL(dest@s_i) =& (\bigcup_{\forall x} DL(\texttt{use}_x@s_i) \cup DL(t_j) \\ =& DL(t_j) \cup (\bigcup_{\forall x. DEF(\texttt{use}_x) \neq \phi} DL(\texttt{use}_x@DEF(\texttt{use}_x)) \\ & \cup (\bigcup_{\forall x. DEF(\texttt{use}_x) = \phi;} \{\texttt{use}_x\})) \end{aligned} \quad (1)$$

As shown by the equations, the lineage set of the variable `dest` that is defined by s_i is the union of the lineage set of t_j and the lineage sets of $\texttt{use}_x$. If a variable $\texttt{use}_x$ was previously defined, $DL(\texttt{use}_x@s_i) = DL(\texttt{use}_x@DEF(\texttt{use}_x))$, otherwise, it is treated as an input and thus $DL(\texttt{use}_x@s_i) = \{\texttt{use}_x\}$.

Table 1 shows the computation of data lineage for the execution trace in Figure 3. In the table, $M[...]$ and P are the abbreviations of $M[...].intensity$ and $P.intensity$, respectively. The last row of the table indicates that the data lineage of $P.intensity$ at 14_0 is computed from the input elements of the original $P.intensity$, $M[0].intensity$, and $M[1].intensity$.

```
1.  i=0;
2.  while (INPUT[i]!=0) {
3.        OUTPUT[i]=INPUT[i];
    }
4.  ...

Execution trace: 1_1 2_1 3_1 2_2 3_2 2_3 ...4_1
```

Figure 4: Effect of Control Dependence.

Control Dependence. Handling control dependence is an important issue. Control dependence is essential to dynamic slicing because a large number of bugs are related to altering the branch outcome of a predicate. However, considering control dependence in data lineage computation may degrade the quality of the results. For example in Figure 4, since each 3_i statement instance in the execution control depends on the corresponding 2_i statement instance, and 2_i control depends on 2_{i-1} since the execution of the ith instance of the `while` statement depends upon the branch

Table 1: Computation of data lineage.

s_i	t_j	def	use_0	$DEF(use_0)$	use_1	$DEF(use_1)$	$DL(def@s_i)/DL(s_i)$
4_0			$M[0]$				$DL(4_0) = \phi$
5_0	4_0	T	P	ϕ			$DL(T@5_0) = DL(P@5_0) \cup DL(4_0) = \{P\}$
6_0	4_0		T	5_0	$M[0]$	ϕ	$DL(6_0) = DL(T@5_0) \cup DL(M[0]@6_0) \cup DL(4_0) = \{P, M[0]\}$
7_0	6_0	P	P	ϕ	$M[0]$	ϕ	$DL(P@7_0) = DL(P@7_0) \cup DL(M[0]@7_0) \cup DL(6_0) = \{P,\ M[0]\}$
4_1			$M[1]$				$DL(4_1) = \phi$
5_1	4_1	T	P	ϕ			$DL(T@5_1) = DL(P@5_1) \cup DL(4_1) = \{P\}$
6_1	4_1		T	5_1	$M[1]$	ϕ	$DL(6_1) = DL(T@5_1) \cup DL(M[1]@6_1) \cup DL(4_1) = \{P, M[1]\}$
10_0	6_1	P	P	7_0	T	5_1	$DL(P@10_0) = DL(P@7_0) \cup DL(T@5_1) \cup DL(6_1) = \{P,\ M[0],\ M[1]\}$
4_2							$DL(4_2) = \phi$
14_0			P	10_0			$DL(14_0) = DL(P@10_0) = \{P,\ M[0],\ M[1]\}$

outcome of the $(i-1)$th instance of the `while` statement. Therefore,

$$\begin{aligned} DL(\texttt{OUTPUT}[i]@3_i)) &= \{\texttt{INPUT[i]}\} \cup DL(2_i) \\ &= \{\texttt{INPUT[i]}\} \cup (\texttt{INPUT[i-1]} \cup DL(2_{i-1}) \\ &= ... \\ &= \{\texttt{INPUT[i]}, \texttt{INPUT[i-1]}, ..., \texttt{INPUT[0]}\} \end{aligned}$$

In other words, even though `OUTPUT[i]` is equivalent to `INPUT[i]`, all the `INPUT`$[x \leq i]$ are considered as being relevant to `OUTPUT[i]`, which is not very useful.

This implies that blindly considering all control dependencies in lineage computation may produce an undesired effect. As it turns out, data dependence is more critical for lineage tracing than control dependence. This claim is borne out by the numerous applications that we have considered from data cleaning, data mining, and scientific applications. It is possible that a data dependence is buried within a control dependence – this is an interesting situation. It is possible to automatically address this case, but the details are beyond the scope of the current paper. In Section 6 we show that for all the programs that we considered data lineage can be correctly computed by considering only data dependencies.

Completeness. We would like to point out that *even tracing both data and control dependencies is not a complete solution*, meaning that relevant input instances may be missing from the lineage set. Consider the example below. Lets assume `INPUT[0]` has the value of 90 such that Statement `3` is not executed. The only statement that Statement 4 depends on is 1. In other words, `OUTPUT`@4 has an empty data lineage set. But we can easily tell that `OUTPUT` is relevant to `INPUT[0]`. The root cause is that the dependence between 2 and 4 is neither a data dependence nor a control dependence, and thus the data lineage set can not be propagated along that dependence. In general, it is hard to capture this type of dependence because of the fact that it manifests itself by not executing certain statements while traditional tracing techniques are only good at capturing what has been executed.

```
1:  OUTPUT = 10;
2:  if (INPUT[0] >100) then
3:      OUTPUT=INPUT[1]
4:  print (OUTPUT)
```

The nature of this type of dependence is very close to that of control dependence and thus it is minor in lineage tracing. This is also confirmed by our experiments, in which we did not encounter any observable problems caused by missing this type of dependencies. Finally, we want to point out that there exist expensive and conservative techniques to compute these *invisible* dependencies [18].

4. IMPLEMENTATION

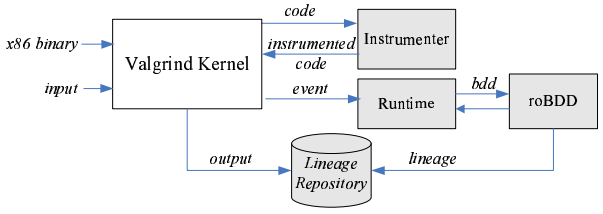


Figure 5: Slicing Infrastructure.

We have implemented the lineage tracing prototype on the tool called *Valgrind*[2] which was originally designed for debugging memory errors in x86 binaries. The kernel of valgrind is a dynamic instrumentation engine which is capable of adding user specified code into the original binary. Therefore, when the original code is executed, the corresponding added code, which is also called *instrumented* code, is executed as well. While previously the instrumentation had the goal of debugging, the valgrind tool can be easily extended by replacing the instrumenter.

Figure 5 illustrates the architecture of our prototype. The *valgrind engine* takes a x86 binary and executes it with the provided input. The engine calls our *instrumenter* when it is about to execute a piece of code. Our instrumenter adds our own code and returns a new piece of *instrumented* code, to the engine to execute. The execution of the instrumented code will result in calling functions provided in the *runtime* component, which perform certain operations based on the semantic of the original code in order to propagate and update the lineage information. The *roBDD* component computes and stores lineage sets. More details about this component are discussed below. Eventually, the system produces both the regular output and the corresponding lineage information. Note that we chose to use valgrind because it is robust and open-source. However, an inherent limitation of valgrind is its speed. Simply executing a program on valgrind without any instrumentation may incur a 10x slowdown. There are industry tools such as *dbt* from Intel and *valcun* from Microsoft, with much lower overhead (as low as 50 percent). Unfortunately, those tools are not publicly available at present.

Next, we discuss two implementation challenges.

The Set Representation. From the earlier discussion, it is clear that lineage computation involves storing a large number of sets and performing set operations at each step of the execution. Therefore, the set representation is critical to performance. A naive link-list based implementation may end up traversing a large set, which may contain thousands of elements, for the execution of a single instruction. Fortunately, recent research on dynamic slicing [27] reveals that *reduced ordered Binary Decision Diagram* (roBDD) [1] can be used to achieve both space and time efficiency in representing sets. RoBDD benefits us in the following respects. Each unique lineage set is represented by a unique integer, which can be considered as an index to the universal roBDD which stores all lineage sets. In other words, two sets are represented by the same integer if and only if they are identical. The use of roBDD achieves space efficiency because it is tailored for set operations characteristic of lineage data such as duplicate removal, and handling overlapping and clustered sets. Set operations can be performed efficiently using roBDDs. More specifically, equivalence tests can be performed in $O(1)$ time [22]. Other binary operations (e.g., union) on two sets whose roBDD representations contain n and m roBDD nodes can be performed in time $O(n \times m)$ [22]. Note that the number of roBDD nodes is often much smaller than the number of elements in the represented set.

Binary Instrumentation. In order to trace lineage, we have to instrument the binary of the program such that lineage information is updated during program execution. According to Equation 1, we need to update the DL set of the left hand side variable at every step of the execution and store it somewhere. In our system, we use *shadow space* to store lineage sets. More specifically, if the variable is stored at a specific stack/heap location, a corresponding *shadow memory (SM)* is allocated and used to store the set associated with the variable. Similarly, we use a *shadow register file (SRF)* to store the sets for variables in registers. Both shadow memory and shadow registers are implemented by software.

```
     register int sum;
1.   A = (int*) malloc (100);
2.   SM(A) = malloc_in_shadow(100);
...
10.  sum = sum + A[i];
11.  SRF(sum) = SRF(sum) ∪ SM(A)[i];
     ...
```

Figure 6: An Example of Instrumentation.

Figure 6 shows an example of instrumentation, the instrumented code is in bold. We can see that an original memory allocation is instrumented with a corresponding memory allocation in the shadow space. An original operation in the program is instrumented with a set operation on the corresponding sets which are stored in the shadow space. *Even though the example is at source code level, the real instrumentation is performed at binary level – without the need to access source code.*

5. STORING FINE-GRAINED LINEAGE

Through the use of our lineage tracing utility, it is now possible to automatically instrument any x86 binary so that it generates fine-grained lineage information for its output. This lineage information can be stored as part of the database in order to make it available for querying.

To record fine-grained lineage, the individual data items must be uniquely identified. If the input is in a flat file, the data items in the file can be identified by the offset in the file and their data length. If the file is in a semi-structured format such as XML, then the scheme proposed in [12] can be used. If the data is in a DBMS, the data item can be identified based on its granularity. The granularity of lineage could be at table, tuple or attribute level. Table level lineage is equivalent to coarse-grained lineage, tuple- and attribute-level lineage are examples of fine-grained lineage.

Our lineage tracing utility provides lineage at attribute level. Tuple level lineage information can be directly computed from the attribute level lineage. The lineage information can itself be stored in a table called Lineage. Table 2 shows an example of how the lineage information can be stored in a database.

id	pid	level	from_id	to_id	Program id
1	318	1	(3,-,-)	(5,-,-)	De-Isotope
2	2122	1	(1,-,-)	(3,-,-)	Data cleaning
3	2122	2	(3,1,-)	(1,101,-)	-
4	318	3	(5,1,5)	(3,5,6)	-
5	318	3	(5,1,5)	(3,15,6)	-

Table 2: Lineage Table

The *id* attribute is the primary key of the lineage table. The *pid* attribute is the identifier of the process that generated the data. The *level* attribute describes the level of the lineage, 1 is table level, 2 is tuple level and 3 is attribute level. *From_id* and *to_id* describe that *from_id* depends on *to_id*. The *program id* attribute stores the id of the program used to generate the derived data. If the input data is in a database, the *from_id* and *to_id* are represented as a triplet *(table_id, tuple_id, attribute_id)*, the first number is the identifier of the table, the second number is the identifier of the tuple in the table and the third number identifies the attribute inside a table. For example, (3,-,-) means Table 3. (5,1,5) means Table 5, Tuple 1, Attribute 5. If the database provides the internal table and tuple identifiers, we could use these as the *tuple_id*. In PostgreSQL, *oid* and *tableoid* columns are created when the table is created. The *oid* uniquely identifies the tuple in a table and *tableoid* identifies the table to which the tuple belongs. The order of attributes in the table can be used as the *attribute_id*. If a key is defined on the table, the key can be used in place of the *oid*. For databases that do not provide the internal tuple identifier, extra tables can be implemented to manage the internal tuple identifier. Further details on how fine-grained lineage can be managed in a RDBMS are discussed in our technical report [26].

6. EXPERIMENTAL RESULTS

In this section, we present an experimental evaluation of the proposed approach using several real applications. Two sets of experiments are presented. To demonstrate the validity of our approach and highlight the importance of fine-grained lineage, we consider the LC-MS application discussed earlier. The second set of experiments is conducted on a range of applications including data mining, data cleansing, and image processing. These experiments are used to study the performance of our technique. The LC-MS application uses real data from a cancer study and

was conducted in collaboration with domain experts. This application is highly sensitive to incorrect (even approximate) lineage and obtaining accurate lineage information is not possible with existing techniques. The experiments below establish the feasibility of our approach and also demonstrate that although there is a distinct slowdown due to lineage tracing, it is not crippling. It should be noted, as mentioned earlier, that for many applications the availability of correct lineage information is far more important than rapid query execution. Even when rapid query processing is necessary, it is possible to compute answers quickly without tracing lineage and later provide lineage information by repeating the query with lineage tracing. In fact, our experiments lead to the improvement of the de-isotoping results and identification of limitations of the current algorithm. These results in themselves were very exciting for our collaborators, which further corroborates the value of having fine-grained lineage (even if it is a little slow to compute).

We use actual mass spectrometer output from a real experiment as input data for our tests. The biological samples were acquired from normal mice and mice bearing breast cancer. The mass difference between normal and cancer labels is 3Da. The code used to process the LC-MS data was obtained from [29]. The experiments are conducted on a machine with 2.40GHz Pentium 4 CPU and 1GB memory running GNU Linux.

6.1 De-isotope Results

As discussed in Section 2 the key goal of biomarker discovery is to identify peptides that show a marked difference in cancer specimens as opposed to normal specimens. Due to the nature of the application, false positives are often produced. These can lead to expensive and fruitless followup research and thus should be eliminated if possible. A key requirement in establishing the validity of a potential biomarker is being able to trace back from the result to the peaks that contribute to this final result. Currently, since fine-grained lineage information is not maintained, this is often done manually and approximately. In the following experiments we show how our technique is able to provide the correct fine-grained lineage in order to rule out false positives from a real experiment. Also, we show that the availability of fine-grained lineage can help identify limitations of the de-isotope algorithm.

6.1.1 Doublet Quantification

Figure 7 (a) shows a portion of a MS spectrum from a real experiment. The de-isotope step identifies 4 peaks, each with charge 4 as shown in Figure 7(b). The peaks in Figure 7(a) are broken up to show their contribution to the final computed peaks in Figure 7(b). These can form two doublets: (P^{σ}, P^{υ}) and (P^{τ}, P^{φ}). However, it turns out that this result is surprising since it implies an unusually large peptide[2]. The availability of fine-grained lineage generated by our method makes it possible to explore this further. The lineage for these peaks is as follows:

[2]Since the charges for these peaks are 4, the peptide that produces the doublet (P^{σ}, P^{υ}) has to contain 3 occurrences of the amino acid Lysine(K) and the peptide that produces the doublet (P^{τ}, P^{φ}) should contain 4 occurrences of Lysine(K). While the occurrence of 3 or 4 Lysine(K) is possible, it is very unlikely.

$$DL(P^{\sigma}) = \{P^{\alpha}, P^{\beta}\}$$
$$DL(P^{\tau}) = \{P^{\alpha}, P^{\gamma}, P^{\delta}, P^{\varepsilon}, P^{\zeta}, P^{\eta}, P^{\theta}, P^{\iota}\}$$
$$DL(P^{\upsilon}) = \{P^{\kappa}, P^{\lambda}, P^{\mu}, P^{\nu}\} \qquad DL(P^{\varphi}) = \{P^{\kappa}, P^{\xi}, P^{o}, P^{\pi}, P^{\rho}, P^{\varsigma}\}$$

From this information, we discover that the m/z difference between the isotopic peaks is approximately 0.33. This implies that the charge should be 3 instead of 4. Obviously there is something wrong. After investigation, we found out that the scientists had inadvertently used an incorrect parameter for the mass accuracy when running the de-isotope function. With the help of the fine-grained lineage we were able to correct this value and set it to a more appropriate value. We ran the function again with the new value.

The new results are shown in Figure 7 (c) and 7 (d), this time the program correctly assigned 4 peaks with charge 3. There are two doublets :($P^{\sigma\prime}$, $P^{\upsilon\prime}$) and ($P^{\tau\prime}$, $P^{\varphi\prime}$). The intensity ratio between $P^{\sigma\prime}$, $P^{\upsilon\prime}$ is 1.45:1, while the intensity ratio between $P^{\tau\prime}$, $P^{\varphi\prime}$ is 1.57:1. These results are promising since the normal ratio is 1:1. Thus, these doublets could potentially be biomarkers of interest. However, before investing more money and effort in investigating these potential biomarkers, it is important to have high confidence that the ratio is correct. The lineage information can once again help establish the confidence in these ratios. In this case, it turns out that the domain experts were satisfied with the lineage. The new lineage information is shown below. In particular, the likelihood that $P^{\sigma'}$ is correct is high since all six isotopic peaks have been identified.

$$DL(P^{\sigma\prime}) = \{P^{\beta'}, P^{\gamma'}, P^{\delta'}, P^{\varepsilon'}, P^{\zeta'}, P^{\eta'}, P^{\theta'}\}$$
$$DL(P^{\tau'}) = \{P^{\iota'}, P^{\kappa'}\}$$
$$DL(P^{\upsilon'}) = \{P^{\iota'}, P^{\lambda'}, P^{o'}, P^{\pi'}, P^{\rho'}\}$$
$$DL(P^{\varphi'}) = \{P^{\iota'}, P^{\lambda'}, P^{\mu'}, P^{\nu'}, P^{\xi'}\}$$

6.1.2 Identifying False Positives

The availability of fine-grained lineage can help improve the quality of the results generated by the de-isotope procedure. Figure 8 shows the results from an experiment that provides an example of this aspect. Figure 8 (a) shows a relatively clean raw spectrum. Figure 8 (b) shows the output of the de-isotope function on this input data. The program detects 4 peaks, which form two doublets (P^{θ}, P^{κ}) and (P^{ι}, P^{λ}). The intensity ratio of doublet (P^{θ}, P^{κ}) is 0.78 and that of doublet (P^{ι}, P^{λ}) is 1.45. The intensity ratio of doublet (P^{θ}, P^{κ}) may be within experiment variation, while doublet (P^{ι}, P^{λ}) could be a potential biomarker. The fine-grained lineage reveals that it is very likely that doublet (P^{ι}, P^{λ}) is a false positive. This is indicated by the fact that peak P^{λ} is not an independent peak, but just a vestige of another peak (P^{κ}) that was produced as a result of the limitations of the de-isotope procedure. This determination is not possible unless we are able to determine the fine-grained lineage. The following is the fine grained lineage of the two doublets in Figure 8 (b).

$$DL(P^{\theta}) = \{P^{\alpha}\} \qquad DL(P^{\iota}) = \{P^{\alpha}, P^{\beta}\}$$
$$DL(P^{\kappa}) = \{P^{\alpha}, P^{\beta}, P^{\delta}, P^{\zeta}, P^{\eta}\} \qquad DL(P^{\lambda}) = \{P^{\alpha}, P^{\beta}, P^{\delta}, P^{\varepsilon}\}$$
$$P^{\lambda} = P^{\varepsilon} - c_0 \cdot (P^{\delta} - c_1' \cdot (P^{\beta} - c_0'' \cdot P^{\alpha}))$$

The main peak P^{ε} in the lineage set of P^{λ}, which is identified by having the same m/z value as P^{λ}, has a much higher intensity than P^{λ}, which is highly unlikely. Further

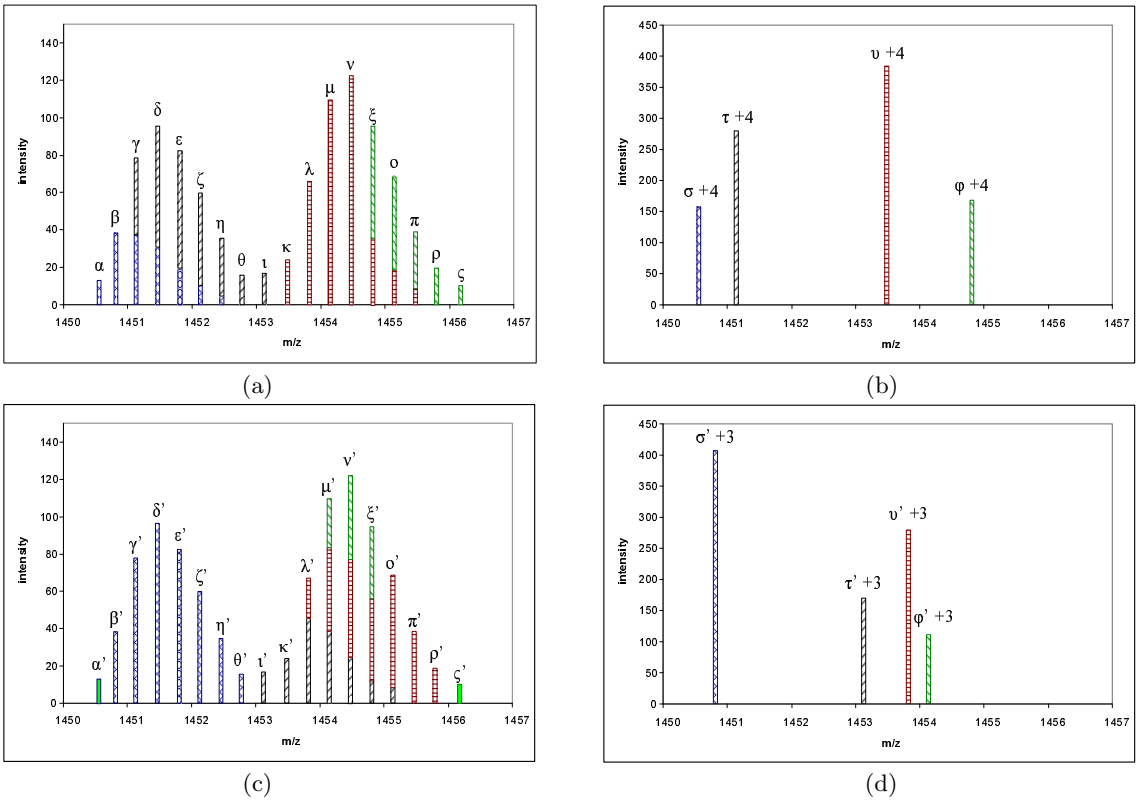


Figure 7: Example showing doublet quantification

inspection showed that the comparison at Line 6 of Algorithm 1 decides that the computation of P^{κ} takes the else branch such that a major portion of intensity is subtracted from P^{ε}, and the remaining intensity constructs the peak of P^{λ}. However, the T value at Line 6 is not necessarily accurate depending on the constant $H[P.MW, i]$, which is calculated by sampling a large database of proteins. The error of the sampling procedure is 5%. This particular peptide very likely falls into this 5%. If we add P^{ι} to P^{θ} and P^{λ} to P^{κ}, then the intensity ratio between P^{θ} to P^{λ} becomes 0.83 which is close enough to the normal ratio of 1:1 to be insignificant. This result itself was very exciting for our colleagues working on biomarker discovery.

Figure 9 shows a more complicated situation where the program was not able to compute the correct answer which was discovered with the help of our fine-grained lineage. Figure 9 (b) is the result of de-isotope. P^{ψ} and P^{ω} are both charge 1 and we infer that P^{ψ} and P^{ω} are a doublet based on their m/z difference. P^{ψ} is the light peptide from normal sample and P^{ω} is the heavy peptide that came from a diseased mouse. After examining the fine-grained lineage information,

$$DL(P^{\psi}) = \{P^{\eta}, P^{\iota}, P^{\lambda}\} \qquad DL(P^{\omega}) = \{P^{\eta}, P^{o}, P^{\rho}, P^{\sigma}\}$$

we are confident that P^{ψ} and P^{ω} are correct and they are indeed a doublet. On the other hand, P^{υ} is suspicious because the program determines its charge to be 2. If we can pair P^{φ} with P^{υ}, they will form a doublet but P^{φ} is charge 1. Note that the value is in m/z, if P^{υ} is charge 2, its molecular weight would have to be 2101 which is far more than 1053.5 that P^{φ} has, therefore P^{υ} and P^{φ} can not be a doublet. We turn to fine-grained lineage for help.

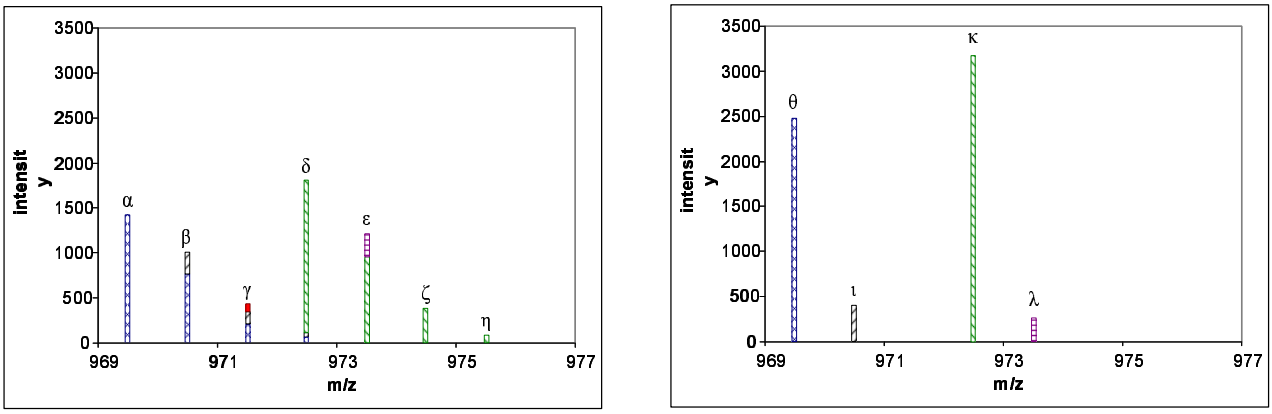
$$DL(P^{\upsilon}) = \{P^{\alpha}, P^{\beta}, P^{\gamma}\} \qquad DL(P^{\varphi}) = \{P^{\eta}, P^{\nu}\}$$
$$DL(P^{\chi}) = \{P^{\theta}, P^{\kappa}\} \qquad DL(P^{\psi}) = \{P^{\eta}, P^{\iota}, P^{\lambda}\}$$

$$P^{\upsilon} = P^{\alpha} + P^{\beta} + P^{\gamma}$$

P^{υ} is determined to be charge 2 because program included P^{β} in its lineage. In fact, the charge of the P^{υ}could be 1 or 2. If the charge of P^{υ} is 1, as shown in Figure 9 (c), P^{Ω} will be assigned charge 1 and appear in the result. P^{υ} and P^{ψ} could pair up and P^{Ω} and P^{χ} could pair up. We will have three doublets. On the other hand, if the charge of P^{υ} is 2, as shown in Figure 9 (d), P^{υ} and P^{χ} pair up. Then we will have two doublets. The program uses heuristics to handle the situation when peptides and their isotopic peaks overlap. In this case, the heuristics fail to produce the correct result. By checking the lineage, we discover the limitations of the heuristics and two alternative interpretations of the raw MS data which are shown in Figures 9 (c) and (d).

6.2 Performance

We selected seven benchmark programs to evaluate the time and space overhead of the lineage tracing technique. `Auto-class` [4] is an unsupervised Bayesian classification system that seeks a maximum posterior probability classification. It takes a database of attribute vectors (cases) as input and produces a set of classes and the partial class memberships as output. The `image` processing program takes a cryo-EM image in tiff format and applies Fourier transformation [16] to the image. The low frequency noise is



(a) (b)

Figure 8: Spectra showing an example of false positive identification.

benchmark	original (sec.)	valgrind (sec.)	tracing (sec.)	tracing/ valgrind
auto-class	0.104	2.92	93.6	32.0
image processing	0.8	5.15	166.3	32.3
lemur	0.85	12.1	302.8	25.0
rainbow	2.22	19.6	286.6	14.6
apriori	2.06	20.7	257.4	12.4
deisotope	9.2	85.8	646.6	7.5
cluto	1.67	42	1670	39.7

Table 3: Running times for benchmark applications.

removed and then another Fourier transformation is performed to covert the image back to a visible form. We used a 512x512 tiff image as input. Lemur [5] is a toolkit designed to facilitate research in language modeling and information retrieval (IR), where IR is broadly interpreted as *ad hoc* and distributed retrieval, structured queries, cross-language IR, summarization, filtering, categorization, and so on. We selected the program `RelEval` from the toolkit to conduct the experiment. This program makes use of the toolkit library and performs 32 feedback queries with pre-constructed index files. Rainbow[3] is a program that performs statistical text classification. It takes documents as input and produces a model containing statistics which can be used to classify documents. The input we used contains 1000 files, each with the size of a few Kbytes. Apriori [7] is a data mining tool which is able to mine association rules. We used a 4 Mbytes input file. De-isotope [29] is the program introduced in Section 2. Cluto [19] is a software package for clustering low- and high-dimensional datasets and for analyzing the characteristics of the various clusters.

In the first experiment, we studied the runtime overhead of the technique. The results are presented in Table 3. The `original` execution times are given in the second column. The column labeled with `valgrind` presents the overhead of the valgrind instrument engine. In other words, we ran the programs on the engine without tracing and collected the execution times. The column with label `tracing` shows the times with lineage tracing on. The last column presents the slow down factor between runs with tracing and without tracing. We chose to compare the execution times between `valgrind` and `tracing` instead of between `tracing` and `original` because valgrind itself often entails x10 slowdown, which undesirably skews the real slow down incurred by the lineage tracing technique.

From the results in the table, we make the following observations.

- The slow down factors range between 7.5-39.8, which we consider as being acceptable in our application domain. The overhead can be easily paid off by the highly valuable lineage information we gain as demonstrated in our case studies.
- The overhead is closely related to the characteristics of a program. For example, in classification type of programs such as `cluto` and `auto-clsss`, individual output values are usually related to a large set of input values, resulting in slow set operations that are involved in lineage computation. `Deisotope` demonstrates the other extreme, in which small lineage sets result in low runtime overhead.
- Part of the runtime overhead is caused by the valgrind engine. As mentioned earlier, replacing valgrind with a more efficient industry-strength instrumentation engine will greatly reduce the overall runtime overhead.

benchmark	orig.(MB)	bdd (MB)	tracing (MB)
auto-class	1.8	1.9	2.2
image processing	16.1	198	16
lemur	14	38.4	9.7
rainbow	6.8	50.8	15.3
apriori	4.1	0.19	3.6
deisotope	125	66.2	17.4
cluto	3	5.2	2.2

Table 4: Memory

Table 4 presents the memory overhead. The original memory usage is presented in the column with label `orig`. The memory overhead stems from two components: the `bdd` component which stores sets and the `tracing` component which propagates lineage sets. The memory consumed by the `bdd` component is mainly decided by the characteristics of the lineage sets. If the sets are repetitive, highly overlapped,

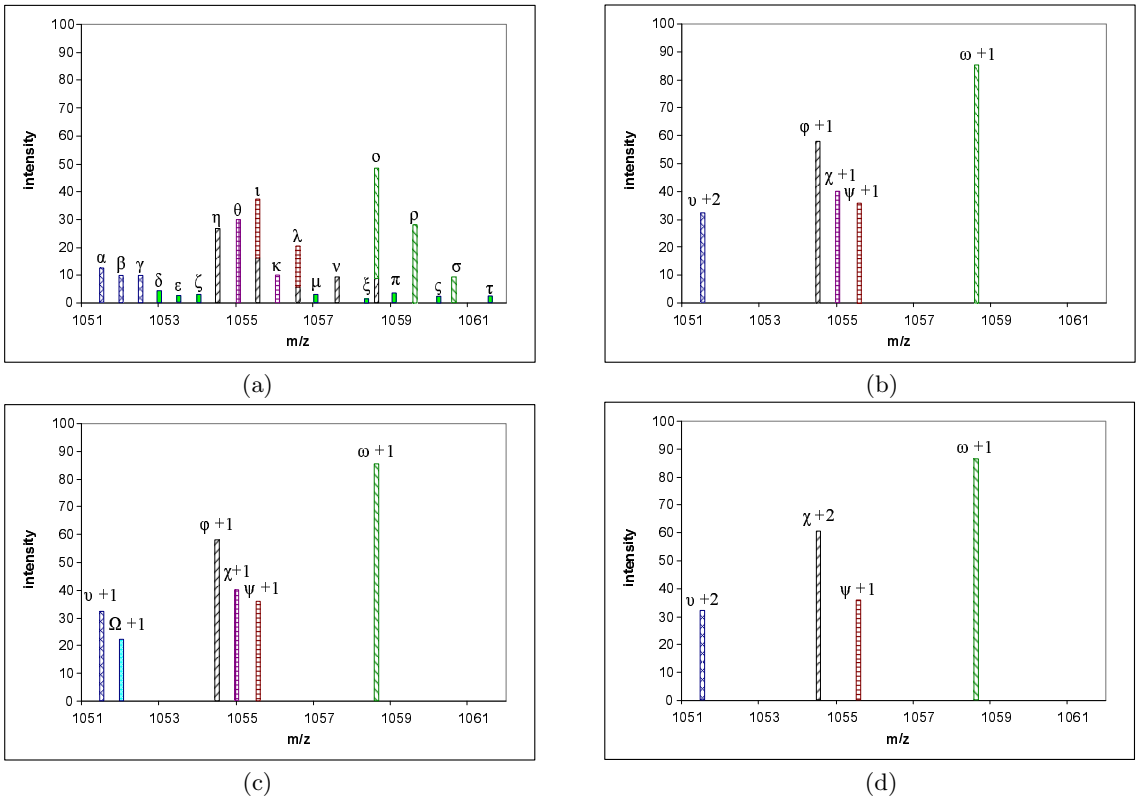


Figure 9: Example spectra highlighting de-isotope function limitations.

or sparse such as in `apriori`, they can be efficiently represented by roBDDs, resulting in less memory consumption. The `tracing` part is mainly decided by the memory footprint of the original execution. As we can observe from Table 4, the memory usages are mostly comparable, which suggests that memory overhead is not the dominant factor compared to runtime overhead.

7. RELATED WORK

Provenance, or lineage, has been extensively studied in the context of scientific computation such as datasets on the grid. One form of provenance is workflow or coarse-grained provenance. For some applications, coarse-grained (i.e., table or file level) lineage is sufficient because typically all elements in the same file or table have undergone the same computational process. Also, the lineage is used to trace the source of abnormality in the data or for data dissemination (i.e., a description of the derivation process is disseminated along with the base data). [11] surveys the use of workflows in scientific computation.

In scientific databases, for example biological databases, coarse-grained lineage is insufficient since not all data values are processed similarly. Although there is a strong need for tracing fine-grained lineage, it remains an unsolved problem. Recently, there has been increasing interest in this area. Cui *et al.* [13] propose fine-grained tracing in the context of data warehousing where all data is produced using relational database queries. The notion of reverse queries that are automatically generated is presented in order to produce all tuples that participated in the computation of a given query. Woodruff and Stonebaker [25] support fine-grained lineage using inverse or weak inverse functions. That is, the dependence of a given result on base data is captured using a mathematical function. They adopt a lazy approach to compute fine-grained lineage upon request from the user. It is not clear if such functions can be identified for a given application. The identification task is highly non-trivial and makes the approach impractical. Similarly, the work on lineage tracing for array data [21] is only applicable in a very limited scenario where the high-level operations are written in Array Manipulation Language (AML).

Buneman *et al.* [12] classified lineage into *why* lineage, which specifies why the data is derived, and *where* lineage, which specifies where the data is copied from. Bhagwat *et al.*[10] proposed three schemes to propagate annotations attached to attributes in relational data.

Dynamic slicing [20] is a debugging technique that captures the executed statements that are involved in computation of a wrong value. Recent research has shown that dynamic slicing is quite effective in locating runtime software errors [28] and dynamic slices can be efficiently computed [27]. The data lineage tracing technique in this paper is based upon the concepts from dynamic slicing such as data/control dependencies. Certain implementation techniques such as roBDD are also reused. The distinction between dynamic slicing and data lineage tracing lies in the information that is traced. In dynamic slicing, a set of executed statements are traced in order to assist programmers in debugging. In contrast, lineage computation traces the set of input that is relevant to a particular output value. A lineage set is usually much smaller than a dynamic slice, which leads to a much more efficient implementation. Fur-

thermore, while control dependence is very crucial in dynamic slicing, it is less important in lineage tracing because data dependence is dominant.

Overall we see that while tracing of fine-grained lineage with non-relational operations is critical for many applications, current solutions fall short of these requirements. To the best of our knowledge, ours is the first work to propose such a system and the only one that can support the types of queries discussed in Section 6 which are of direct relevance to scientists.

8. CONCLUSIONS

Fine-grained lineage is extremely valuable for many scientific and database applications. Several efforts have been focussed on computing fine-grained lineage when relational operators are used to transform data. However this work is not applicable to the important and common case for many applications that employ non-relational operators for processing data. The current work for arbitrary operators is very limited and cannot be applied automatically without specific domain knowledge. In this paper, we presented the first technique that can trace fine-grained lineage across arbitrary operators. Our method is motivated by the technique of dynamic slicing used for debugging. We have shown how this technique can be modified for tracing fine-grained lineage. The new technique was shown to be accurate and highly effective using a real application for scientific data. The technique has the significant advantage that it does not require any domain knowledge, access to source code, or human intervention. The results from our method were shown to help improve the accuracy and reliability of the de-isotope function. In addition, we showed that although the tracing method does introduce a significant slow down, it is still an extremely valuable tool. This is especially true for scientific applications for which the availability of fine-grained lineage is a major obstacle. Thus making this data available is far more important than the extra time needed to compute it. Moreover, since this represents the first step in developing such a system, we expect that future work will help improve the runtime cost of the tracing. We tested the method on a wide variety of applications and showed that tracing is easily achieved in each case. The new technique is useful for any application that uses complex processing and requires lineage tracing such as drill through operations in data warehouses and data cleansing.

9. REFERENCES

[1] Buddy, a binary decision diagram package. Department of Information Technology, Technical University of Denmark.

[2] http://valgrind.org.

[3] http://www.cs.cmu.edu/ mccallum/bow.

[4] http://www.cs.purdue.edu/homes/mgelfeky/dq/.

[5] http://www.lemurproject.org/.

[6] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[7] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[8] G. Alonso and C. Hagen. Geo-opera: Workflow concepts for spatial processes. In *Symposium on Large Spatial Databases*, pages 238–258, 1997.

[9] A. Beszedes, T. Gergely, Z. M. Szabo, J. C., and T. Gyimothy. Dynamic slicing method for maintenance of large c programs. In *CSMR*, 2001.

[10] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[11] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[12] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[13] Y. Cui and J. Widom. Lineage tracing in a data warehousing system. In *ICDE*, pages 683–684, 2000.

[14] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. The virtual data grid: A new model and architecture for data-intensive collaboration. In *CIDR*, 2003.

[15] I. T. Foster, J. S. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM*, 2002.

[16] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[17] P. Groth, S. Miles, W. Fang, S. C. Wong, K. P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *HPDC'05*, July 2005.

[18] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In *ESEC/FSE-7*, pages 303–321, 1999.

[19] George Karypis. Cluto - a clustering toolkit. Technical Report 02-017, Computer Science and Engineering, University of Minnesota, April 2002.

[20] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[21] A. P. Marathe. Tracing lineage of array data. *J. Intell. Inf. Syst.*, 17(2-3):193–214, 2001.

[22] C. Meinel and T. Theobald. Algorithms and data structures in vlsi design, 1998. Springer.

[23] S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*, 2006.

[24] R. D. Stevens, A. J. Robinson, and C. A. Goble. mygrid: personalised bioinformatics on the information grid. *Bioinformatics*, 19(Suppl 1):i302–i304, 2003.

[25] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.

[26] M. Zhang, D. Kihara, and S. Prabhakar. Tracing lineage in multi-version scientific databases. Technical Report CSD TR 06-013, CS, Purdue University, July 2006.

[27] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE*, 2004.

[28] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, 2005.

[29] X. Zhang, W. Hines, J. Adamec, J. Asara, S. Naylor, and F. E. Regnier. An automated method for the analysis of stable isotope labeling data for proteomics. *J. Am. Soc. Mass Spectrom*, 16:1181–1191, 2005.

[30] W. Zhu, X. Wang, Y. Ma, M. Rao, and J. S. Glimm, J.and Kovach. Detection of cancer-specific markers amid massive mass spectral data. *Proc Natl Acad Sci U S A*, 100:14666–14671, 2003.