Tracing Lineage Beyond Relational Operators

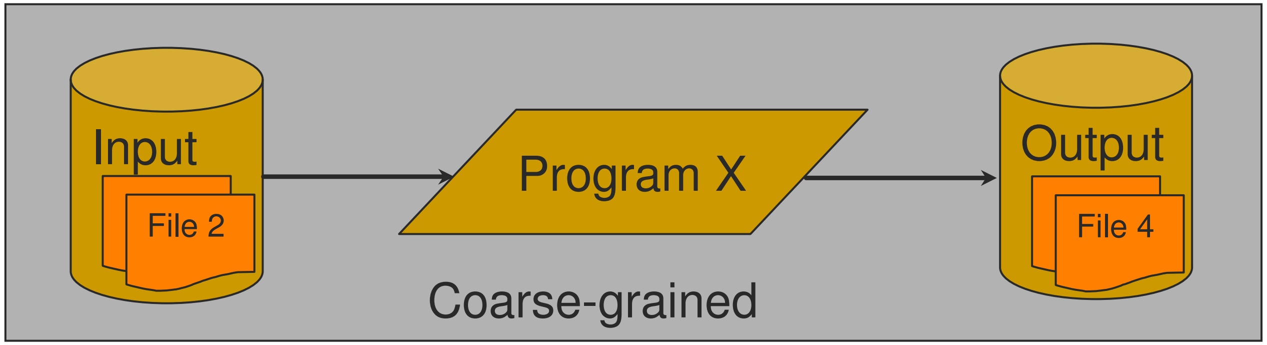
Mingwu Zhang[1] Xiangyu Zhang[1]
Xiang Zhang[2] Sunil Prabhakar[1]

[1]Computer Science
[2]Bindley Bioscience Center
Purdue University

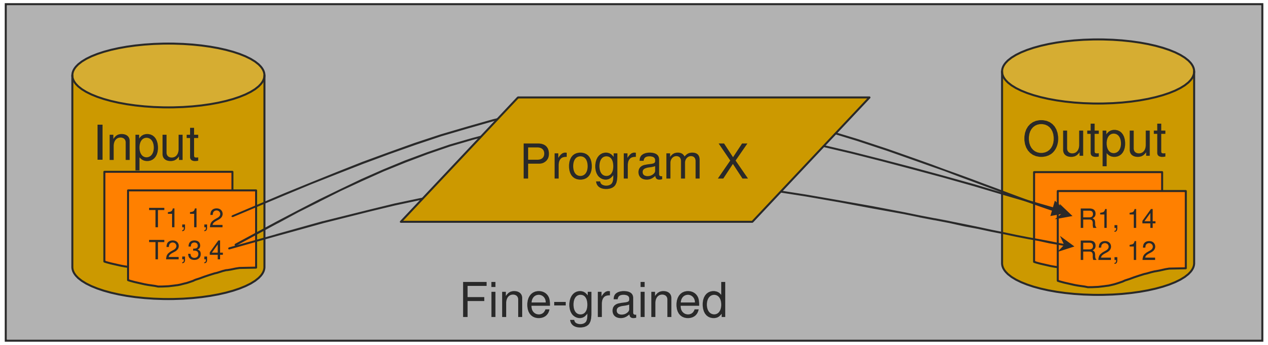
Introduction

- Lineage (Data Provenance) is defined as description of the origin of the data and the process by which the data is derived.
- Lineage is
  - critical for determining data quality and reliability (e.g. biological data, data cleansing)
  - mandated by law (e.g. audit trails for FDA)
  - essential for data dissemination and reproduction
  - Informative (e.g. querying lineage)
- Database support for tracing lineage is urgent

Lineage tracing

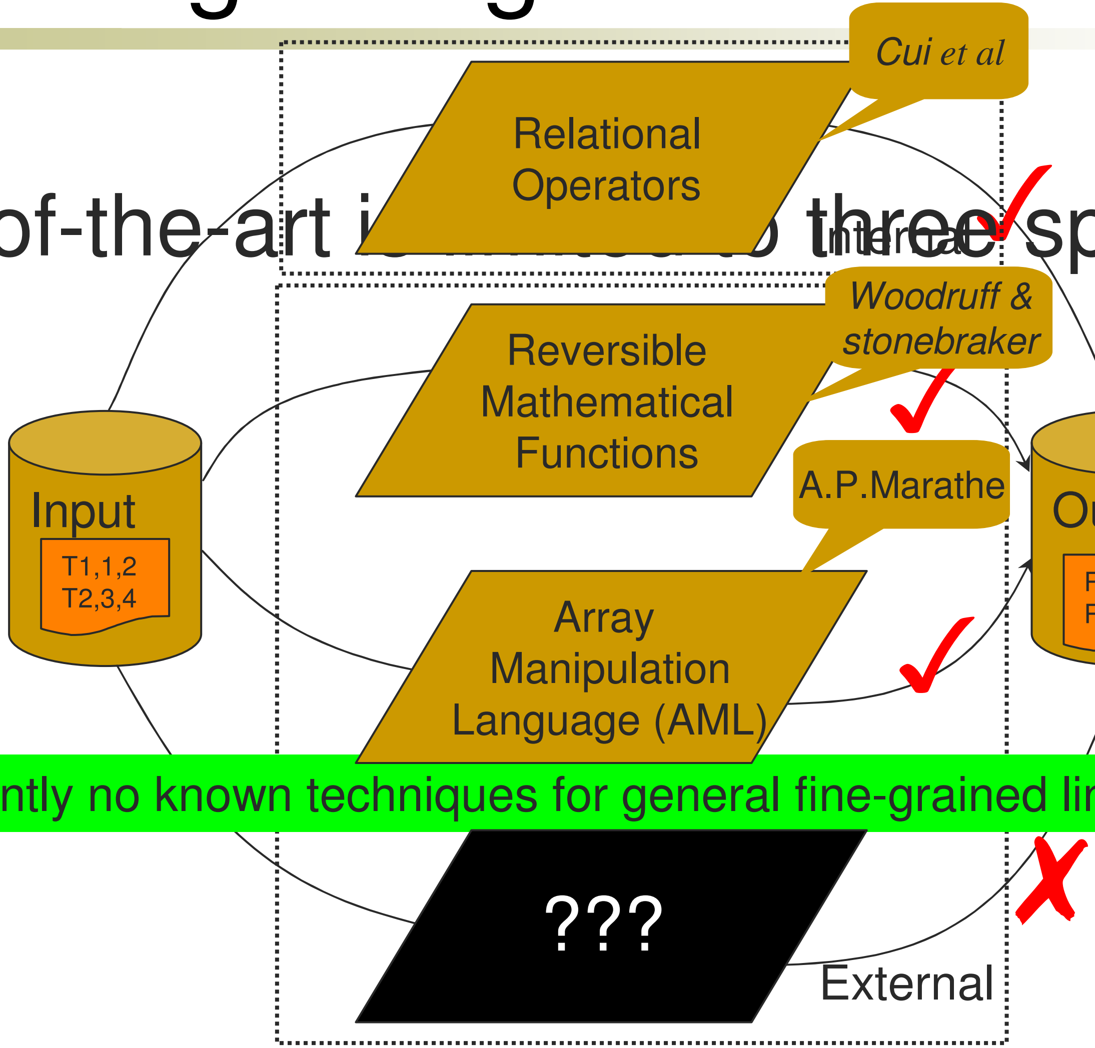


e.g Workflow level lineage



Tracing fine-grained lineage

State-of-the-art is limited to three special cases



Cui et al

Relational Operators

Internal

Woodruff & stonebraker

Reversible Mathematical Functions

A.P.Marathe

Input

T1,1,2
T2,3,4

Output

R1,12
R2, 34

Array Manipulation Language (AML)

???

External

✓ ✓ ✓ ✗

Currently no known techniques for general fine-grained lineage tracing

Contributions

- Enable fine-grained external lineage tracing for any arbitrary program without requiring
  - Domain expertise
  - Understanding the semantics of the operation
  - Source code
- Computed lineage is accurate (no false positives)
- Lineage is derived directly from program execution.

Outline

- Introduction
- Lineage tracing
- Case study
- Conclusion

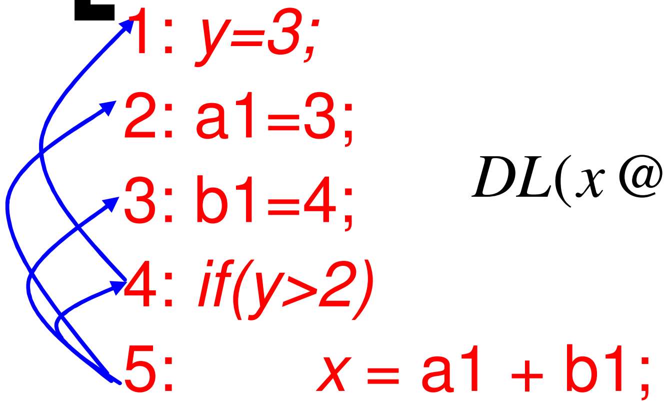
Our approach

- Automatically trace lineage using only binary executables.
- Monitor the data flow during program execution.
- As the program is executed, the lineage of each variable is traced.
- Each binary instruction is modified to keep track of the data/control dependencies generated by the instruction.

Tracing Lineage

- A statement *S* **data depends** on another statement *t* if and only if a variable is defined at *t* and then used at *S* .
- A statement *S* **control depends** on a predicate statement *t* if and only if the execution of *S* is the result of the branch outcome of *t*
- *Definition*: Given a program execution, the data lineage of variable *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 the computation of the value *v* at S_i

Tracing example



```
1: y=3;
2: a1=3;
3: b1=4;
4: if(y>2)
5:        x = a1 + b1;
```

$$
\begin{aligned}
DL(x@5) &= (DL(a1@5)\,\mathrm{Y}\,DL(b1@5))\,\mathrm{Y}\,DL(4)\\
&= DL(a1@2)\,\mathrm{Y}\,DL(b1@3)\,\mathrm{Y}\,DL(y@4)\\
&= DL(a1@2)\,\mathrm{Y}\,DL(b1@3)\,\mathrm{Y}\,DL(y@1)
\end{aligned}
$$

- At Statement 5, x depends upon a1,b1 (data dependency) and y (control dependence)
- Thus, the lineage of x is the union of the lineages of a1, b1 and y

Deriving lineage

Let s_i : *dest* = ? t_j: *f*(use_0, use_1, …, use_n)

be the executed statement instance s_i, which assigns a value to variable *dest* by using variables use_0, use_1, …, use_n and s_i control depends on t_j. Let *DEF(x)* be the latest statement instance that defines x.

$$DL(dest @ s_i) = (\bigcup_{\forall x} DL(use_x @ s_i)) \cup DL(t_j)$$

$$= DL(t_j) \cup (\bigcup_{\forall x . DEF(use_x) \neq \phi} DL(use_x @ DEF(use_x))$$

$$\cup (\bigcup_{\forall x . DEF(use_x) = \phi} \{use_x\}))$$

Instrumenting the code

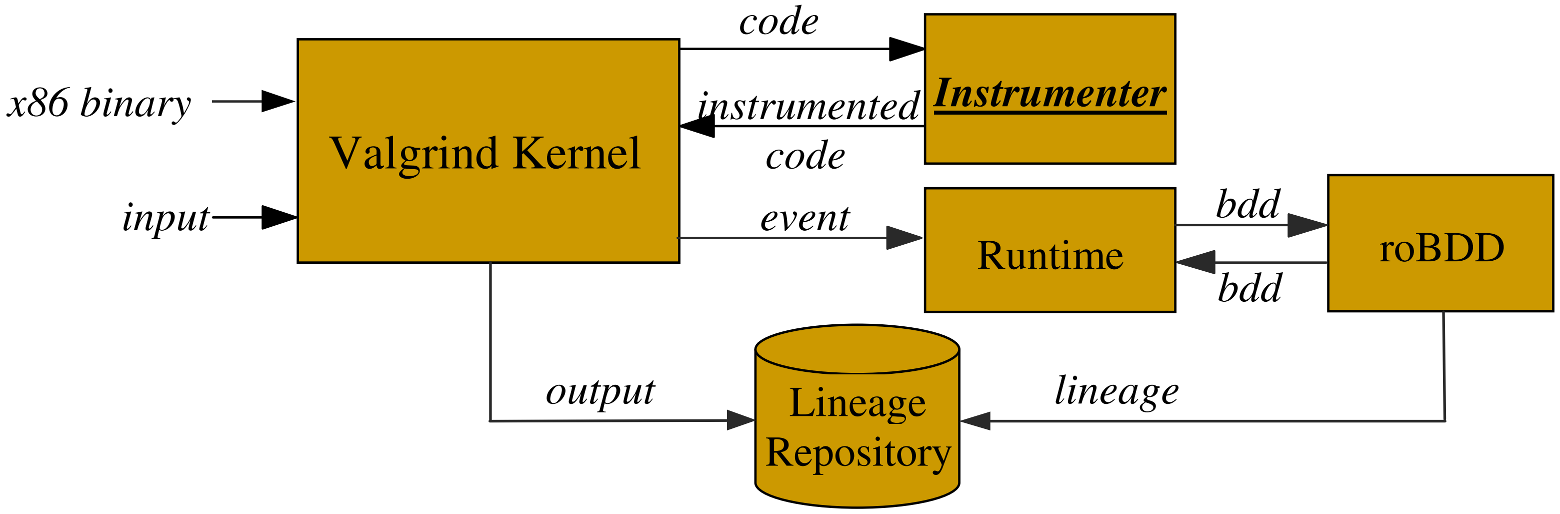
- We use an open-source instrumentation kernel called Valgrind
- The lineage is typically set data and is stored using a structure called roBDD which is optimized for set operations.
- Shadow memory (SM) stores the lineage sets associated with variables in stack/heap and shadow register file (SRF) stores lineage sets for variables in registers

Instrumenting the code

- Example instrumentation
  - A=(int*) malloc(100)→
SM(A) = malloc_in_shadow(100)
  - add (0x0884dc0), eax→
mov SM[0x0884dc0] U SRF(eax), SRF(eax)

Architecture

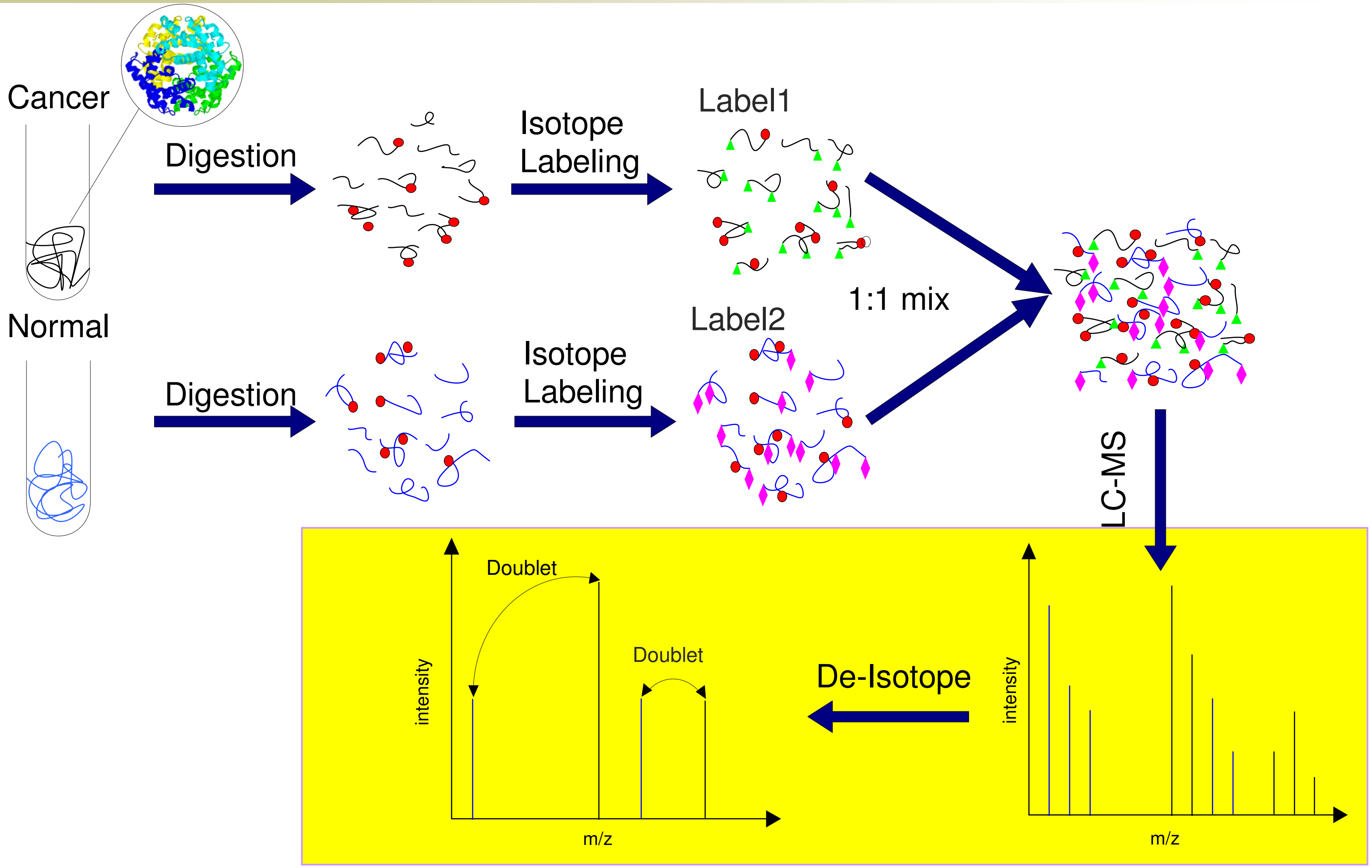
- We have developed a prototype system based on Valgrind engine to trace fine-grained lineage



Outline

- Introduction
- Lineage tracing
- Case study
- Conclusion

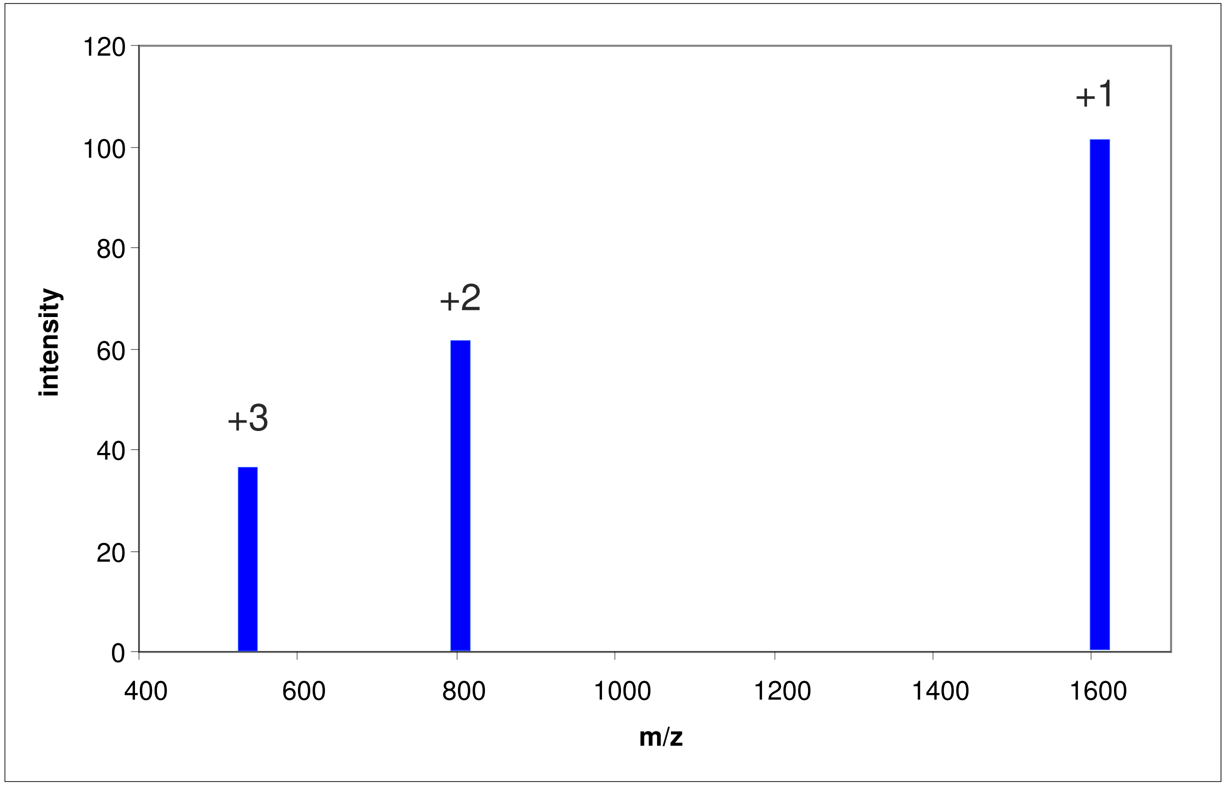
Case study: Cancer biomarker discovery



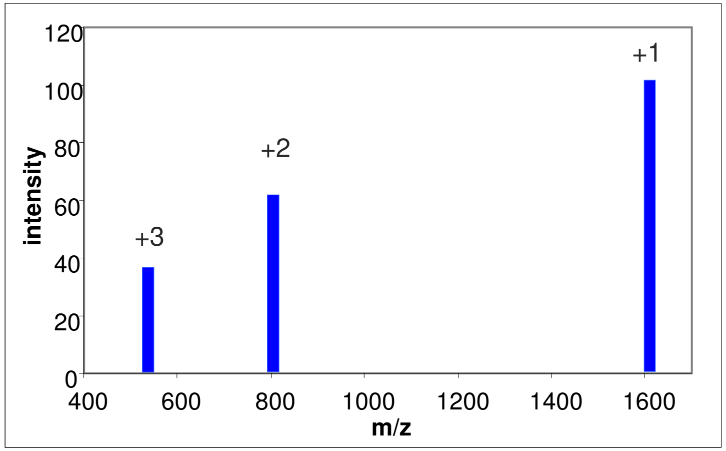
Mingwu Zhang et al.

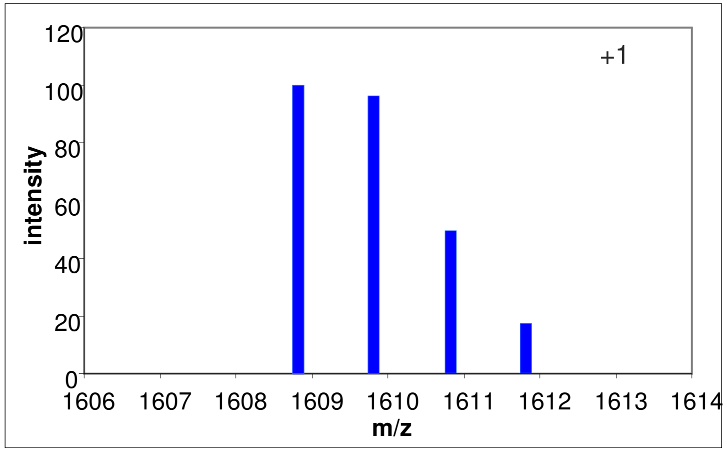
De-isotoping

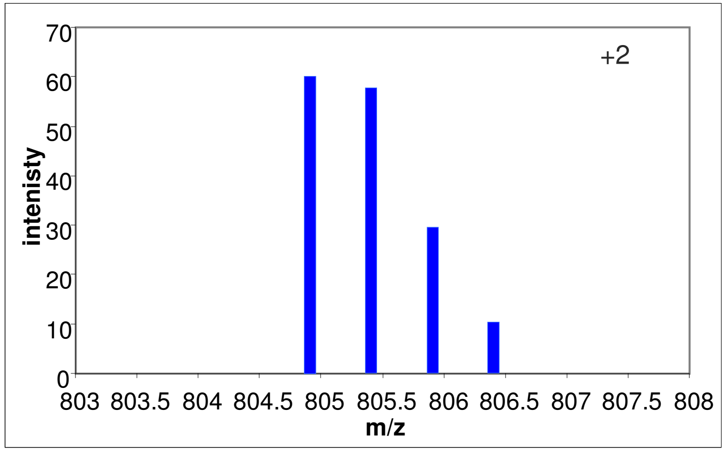
Seq1: ATLNELVEYVSTNR

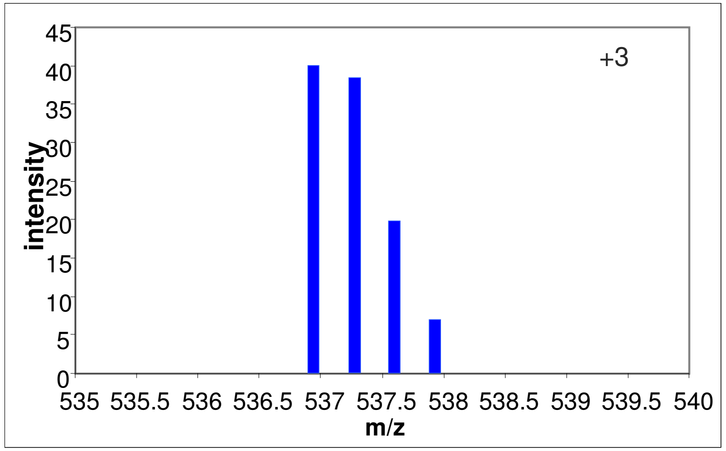


De-isotoping





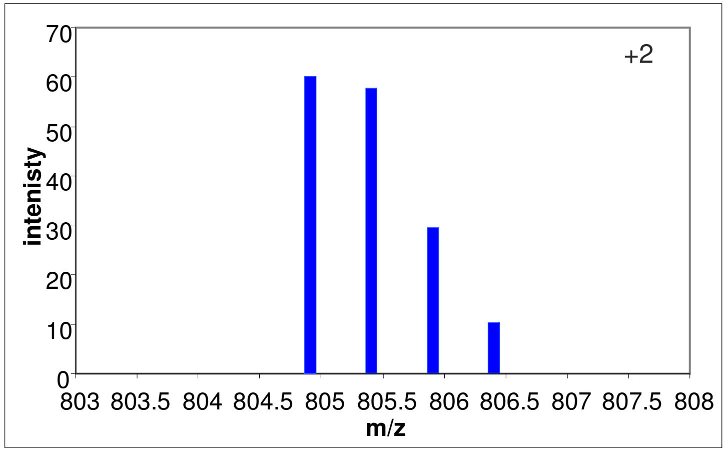


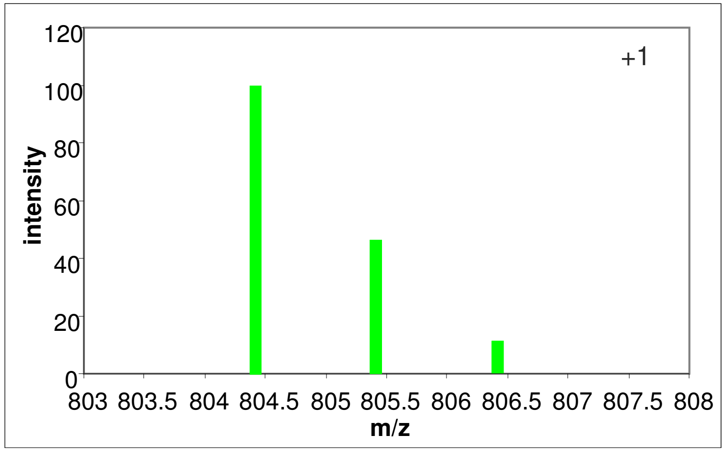


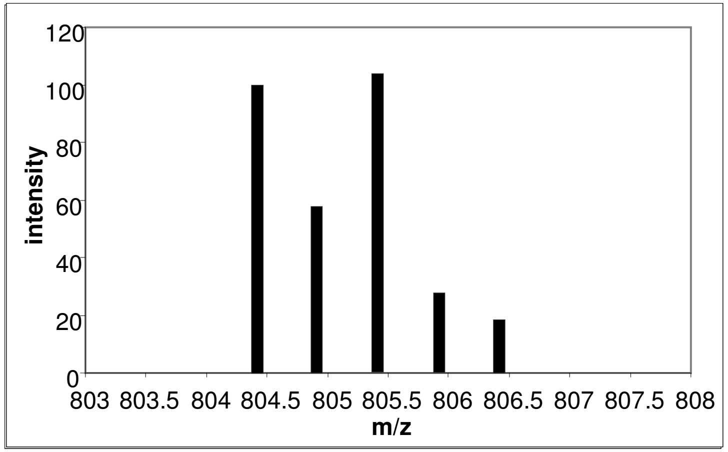
De-isotoping

Seq1: ATLNELVEYVSTNR

Seq2: ITCAELR





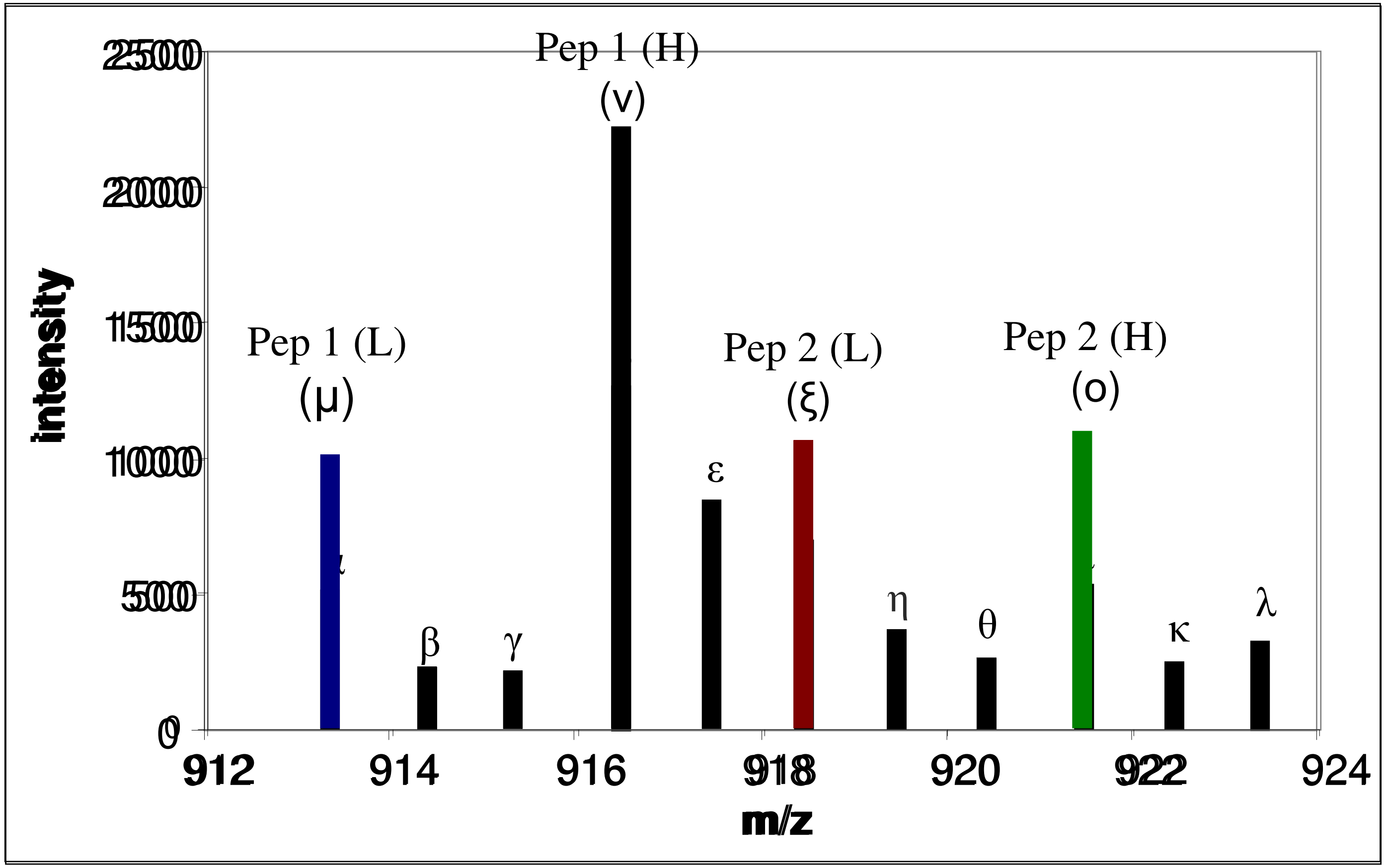


Mingwu Zhang et al.

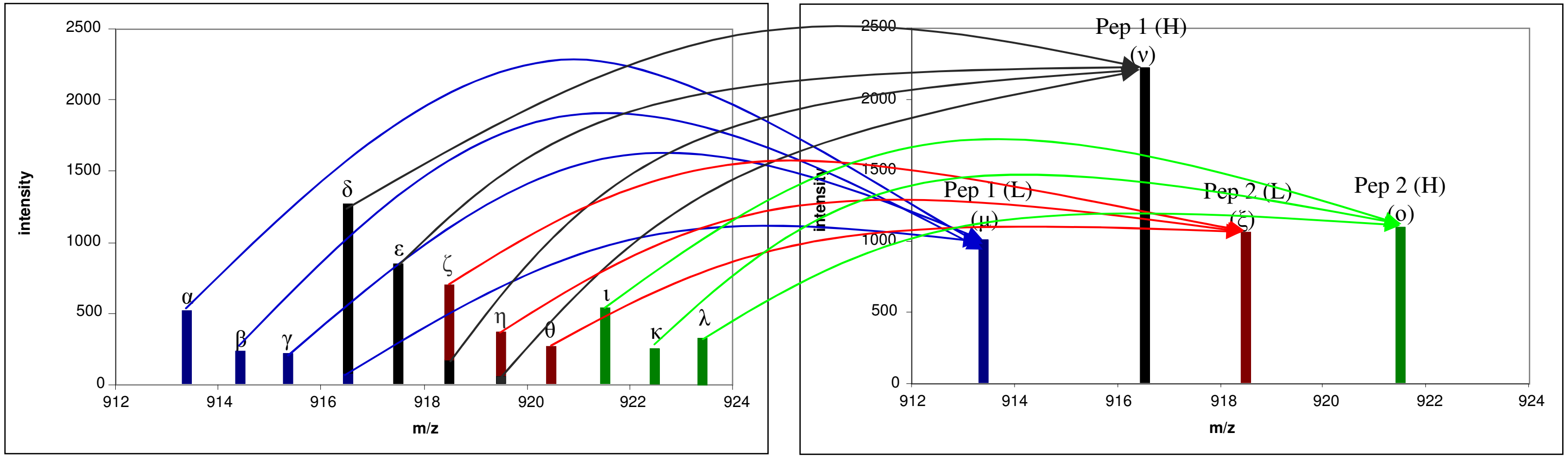
De-isotope algorithm

- Complex, mostly heuristics
- Numerous parameters picked by experts
- Validity of results can be affected by choice of parameters
- Identifying a reverse function is impossible, even for experts.
- Using state-of-the-art algorithm.

De-isotope result



Fine-grained lineage

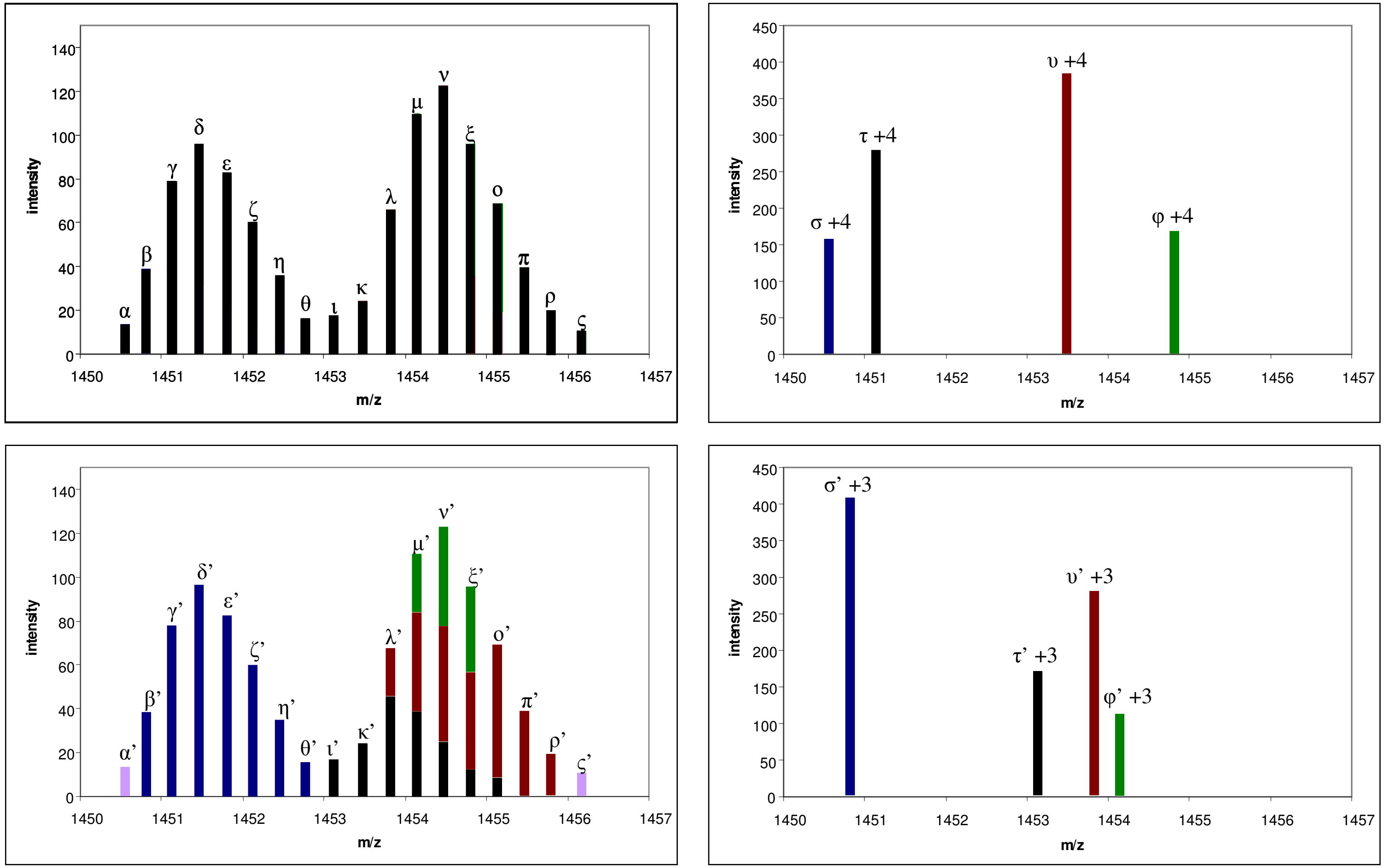


Mingwu Zhang et al.

Case study

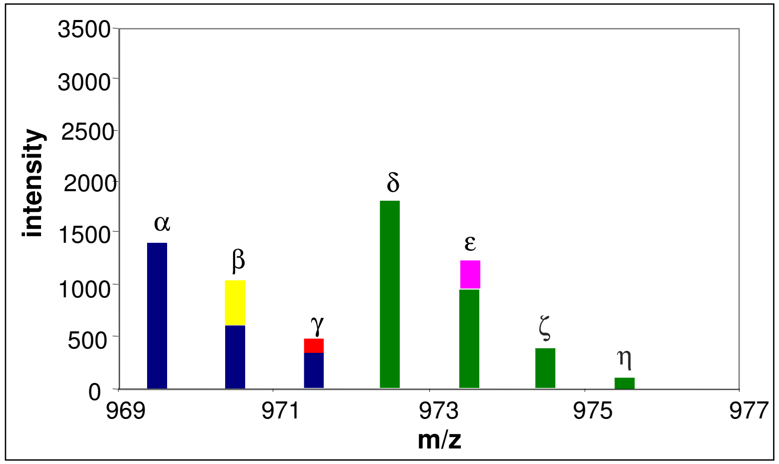
- External fine-grained lineage is crucial for our biomarker discovery application
- Our technique enabled experts to
  - detect errors
  - assess the reliability of data
  - identify false positives
  - identify program limitations

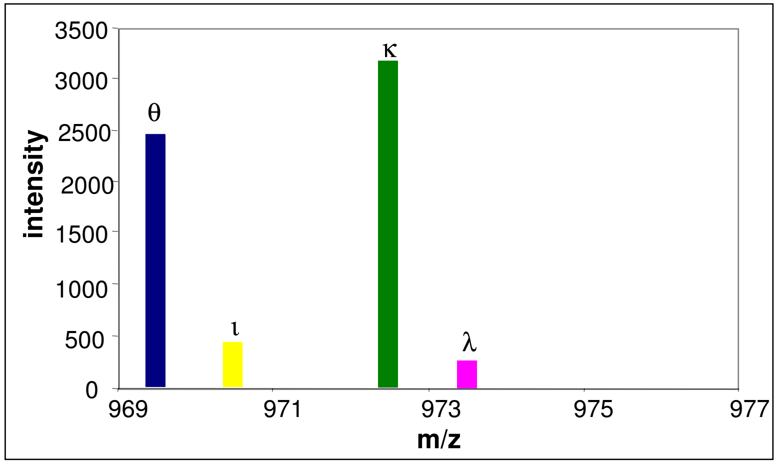
Detect error



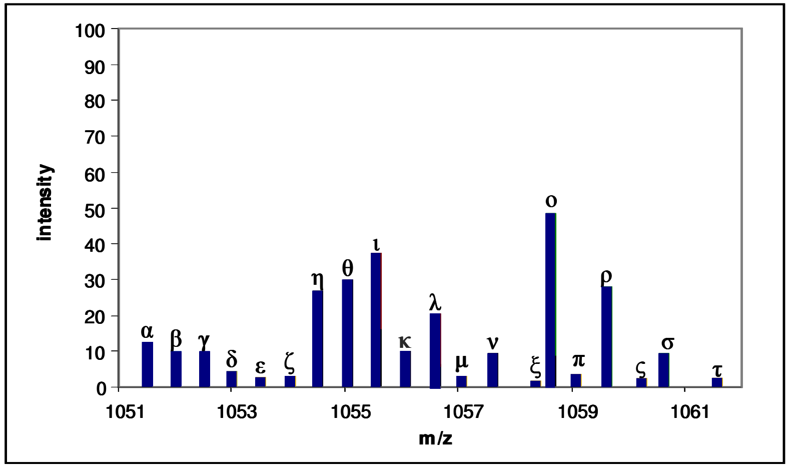
Mingwu Zhang et al.

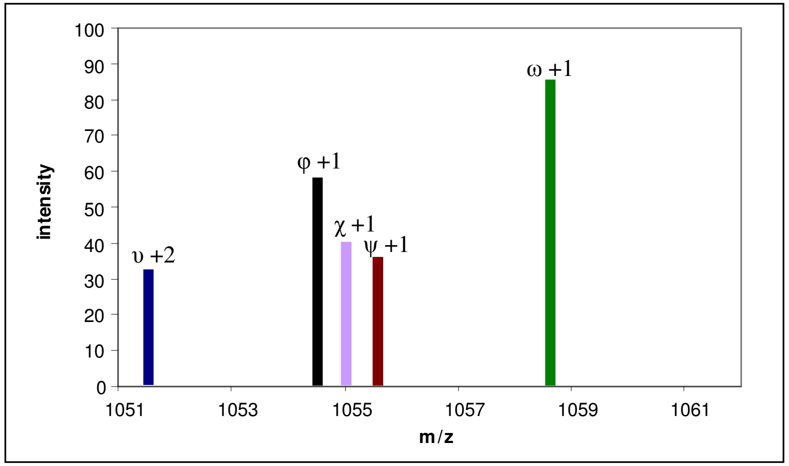
Identifying false positives

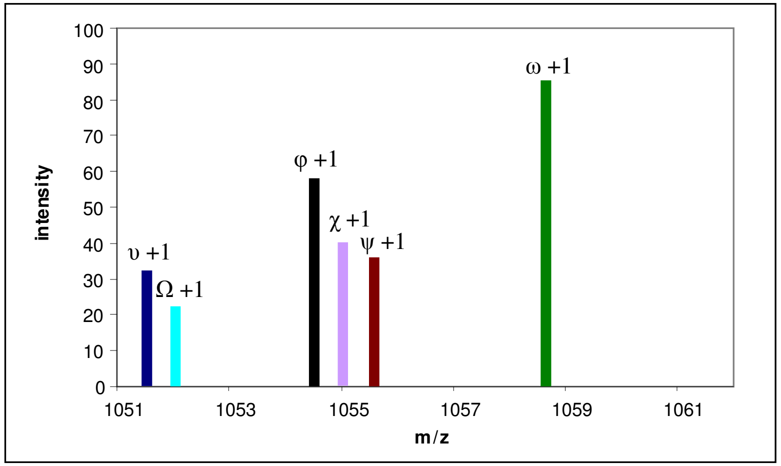


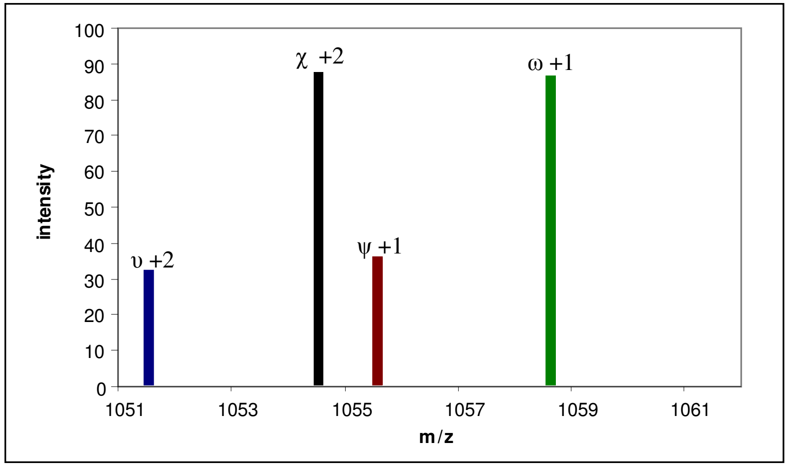


Program limitations









Performance

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

Memory consumption

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

Outline

- Introduction
- Tracing lineage
- Case study
- Conclusion

Conclusion

- This is the first work that can trace external general fine-grained lineage
- Advantages
  - Fully automated
  - Does not require user input or domain knowledge
  - Does not need source code
  - The lineage is accurate

Conclusion

- Disadvantage
  - Performance
    - Tracing lineage incurs a slowdown but acceptable for applications in need of lineage
    - Part of the overhead is caused by Valgrind, other industrial instrumentation engines such as *dbt (Intel)* and *valcun (Mirosoft)* incur less overhead.