Runtime Analysis of Distributed Data Processing Programs

Marcus Leich
supervised by Prof. Volker Markl
Technische Universität Berlin
Einsteinufer 17
10587 Berlin, Germany
firstname.lastname@tu-berlin.de

ABSTRACT

Debugging massively parallel data analysis programs is currently a difficult process. Traditional debug cycles involve manual code instrumentations, re-execution and analysis of the resulting data. This is expensive in terms of development time, execution time, amount of data produced, and cognitive overhead. This work proposes a course of research that is meant to alleviate this situation by automating the code instrumentation and by lowering the re-execution time of instrumented code. By using these techniques, we hope to achieve a higher efficiency compared to manual debugging approaches.

1. INTRODUCTION

Implementing and testing massively parallel data analysis programs (MPDAP)s is a tedious process. In current systems, such as Hadoop MapReduce, Stratosphere, and Spark, developers usually implement user defined functions (UDF)s. These functions are passed as arguments to nested second order functions that represent data flow graphs[1]. Like any other piece of software, such programs, especially the UDFs, are bound to contain bugs. Compared to debugging on single machines, finding bugs in distributed shared-nothing environments with hundreds of nodes that process terrabytes of data is more difficult. The reason for this is that the most commonly used debugging strategies, *print* statements and interactive break-/watchpoint debuggers, in combination with cyclic re-execution, do not scale well to massively parallel environments.

Firstly, execution times are often not developer friendly. A program that runs several hours is likely to cause similarly long debug cycles. Even if execution times are within developer friendly scales, short debug cycles are still costly in terms of computational resources. Effectively, a five minute job that saturates a 100-node cluster, still consumes several CPU hours that could be put to better use.

[1]Out of the box Hadoop MapReduce only allows fixed graphs with one map and one reduce operator.

Proceedings of the VLDB 2014 PhD Workshop.

Secondly, the amount of log and tracing data that is produced during the individual debugging cycles can itself represent a big data problem that needs to be solved. When analyzing a certain failure, developers need to have a good understanding of what input and intermediate data looks like. Even in conventional small scale applications, simply inspecting a log of all values that occurred during execution may already be infeasible. In the context of big data, the problem size increases by several orders of magnitude. Here, materialization, let alone inspection, of potentially billions of input or intermediate values is out of question. The only valid solution appears to be gathering meaningful statistics for a given debug task. Writing the corresponding statistics gathering and aggregation logic is just as error-prone as writing the original program that is to be debugged.

Of course, well-known quality assurance techniques may help to intercept potential bugs before execution on the entire dataset. Locally executed unit tests for individual code modules, and integration tests for entire analysis programs may help to test and debug code locally before running it on the cluster. However, a lot of data analysis programs, especially those used during exploratory analysis, may combine code modules in unforeseen ways and thus provoke erroneous code interactions. In terms of data size, it is of course possible to sample the input dataset(s) for local execution. However, the sampled dataset is likely to not accurately represent all peculiarities of the original dataset.

We argue that all of these techniques fail to account for the ad-hoc nature of many data processing programs as well as for a specific property that holds for all but the most cleanest data sets: *failure waits in the long tail.* According to the law of large numbers, even the most unlikely error will occur eventually if only the data set is large enough. Consequently, we see the need to enable developers to debug their MPDAPs *in situ*, that is in its native environment, the cluster, and with the entire dataset.

It is our goal to research means for minimizing the inevitable debugging costs of MPDAPs running in shared-nothing environments. Specifically, we focus on minimizing re-execution times. The core problem that causes costly re-executions is that unintended program behavior often cannot be readily tracked back to it's root causes: unexpected input data, erroneous code, or a combination of both. We intend to solve this problem using a combination of the following techniques:

1. Automated Instrumentation (AI) of MPDAPs

2. Cross UDF Slicing (CUS) of MPDAPs

3. Data Slicing (DS) in MPDAPs

The following section provides details on the three core techniques that we propose in this work. Section 3 and 4 sketch our implementation and validation strategies. Section 5 describes the related work and Section 6 outlines our work plan for the following years.

2. TECHNIQUES

2.1 Automated Instrumentation

Instrumentation is the process of supplementing the data analysis program or any of its underlying execution systems (e.g. Hadoop MapReduce, Stratosphere, the Java Virtual Machine) with code that gathers intelligence such as values of tuples, aggregations, like number of input/output tuples of an operator, event counters, or statistical distributions.

In current systems, this is usually done manually by developers by adding suitable logging functionality to their code. After discovering a faulty result, developers add code to log properties[2] of contributing variables. Similarly, developers may introduce counters to simply log how often certain code paths are triggered or how often boolean expressions such as equalities, inequalities, and *null*ness hold.

Since these modifications are time consuming and error prone, there is an incentive for programmers to keep instrumentation as low as possible. Consequently, developers may have to repeat this process several times to find the root cause of a faulty result.

We propose to automate this process and thus make it less error prone. Developers should be enabled to simply specify code path locations, variable identifiers, or boolean conditions and have the necessary instrumentation performed for them. During subsequent executions of the program, the gathered data should be aggregated across the cluster and delivered to the developer's workstation. This should reduce the number of execution cycles, since developers would not be restricted by implementation overhead when specifying desired instrumentation points. Given proper implementation, automated instrumentation will also reduce the likelihood of introducing new bugs during instrumentation.

2.2 Cross UDF Slicing

Program slicing is a well-known debugging technique that isolates program statements that potentially contribute to a faulty result, hiding the clutter of surrounding, non-contributing code [20]. The analysis techniques are very similar to read/write set analysis as it has been applied in [12].

To enable program slicing for MPDAPs, we need to be able to determine the contributing statements of a result across UDFs, thus following the origins of a given result back to the data sources. Since the semantics of the individual operators in the operator graph are well understood, it should be possible to extend the concept of program slicing to this type of program.

The benefit of this approach is threefold:

1. In combination with an IDE, it would be possible to provide developers with direct visual cues that indicate which statements, across the entire program, contribute to the values of any given variable. This feature would allow for sanity checks even without executing the program.

2. CUS could provide possible instrumentation points to the AI algorithm. In this scenario, the developer would only need to select the variable that contains the faulty result and CUS would determine the code locations relevant for AI.

3. Once a variable containing faulty results has been identified, CUS could be used to automatically strip any non-contributing code from the program and thus reduce it to the critical path. Depending on the ratio and complexity of non-contributing code, this could significantly reduce re-execution time.

[2]min/max values, histograms, or just samples

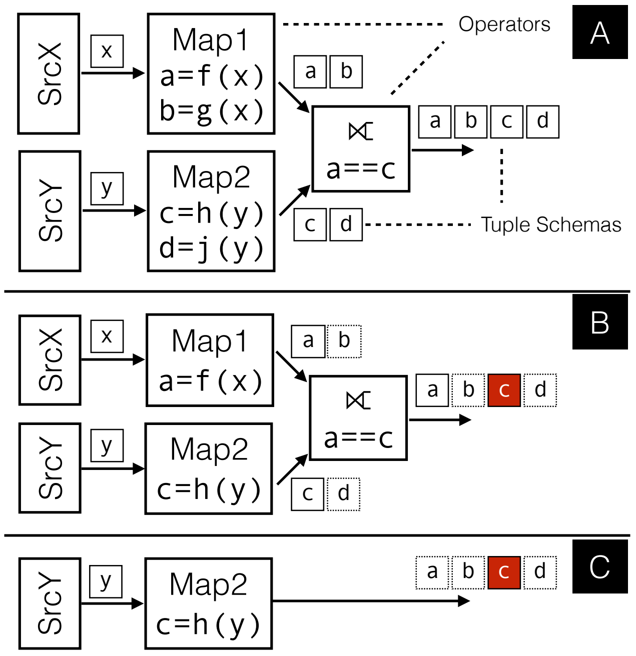

Figure 1: Illustration of Cross UDF Slicing. A: Original data flow. B: Sliced data flow that correctly computes all values of c (dashed tuple components are not computed). C: Sliced data flow that omits duplicate values of c that are introduced by the right outer join.

Especially point three is closely related to the idea of query or program optimization. The removal of non-contributing operations is a standard feature of such optimizers. Consequently, it would be possible to use an existing query optimizer of the execution environment to prune any non-relevant statements within UDFs and even entire data-flow subgraphs. It should be noted though, that decomposing (and potentially recombining) UDFs is a highly advanced optimization feature that, to the best of our knowledge, has not been implemented in query optimizers of current systems. The strived for results of this work can therefore be seen as enabling technology for advanced query optimization.

Figure 1 illustrates the concept using a simple program with two sources, two map operations, and a right outer join. The sliced version of this program (Figure 1B) that correctly computes all values of c still includes all operators, but leaves out computations for the irrelevant values b and d. Due to the right outer join, the sliced program still needs to compute a completely, because multiple occurrences of certain values of a may also cause multiple occurrences of c. If we relax conditions slightly and assume we can omit duplicate values that are introduced by the join, the sliced program can be reduced to only containing a sliced version of *Map2*. Depending on the complexity of the pruned operations, sliced programs may exhibit significant performance speed-ups.

A challenge for program slicing are third-party libraries and system code. These components are usually not in the debugging scope of a developer, since they are either unlikely to contain bugs or their implementation details do not concern the developer. In this case, program slicing should still be able to follow the dependencies of any given variable trough the system code. However, due to licensing constraints, the code itself may not actually be sliced.

2.3 Data Slicing

Building on top of automatic instrumentation and on the idea of program slicing, we propose data slicing as a third approach to reducing re-execution time. Here, the idea is to reduce execution time not by removing irrelevant parts of the program logic, but by ignoring irrelevant parts of the data. This strategy borrows from the concept of *data provenance* [2] although exact provenance information is not always required during debugging. For reducing re-execution time, it can be sufficient to have approximate provenance information. This approach assumes that the program can be executed multiple times on precisely the same dataset.

As an example we assume a single faulty result in a groupwise aggregation as depicted in Figure 2A. In order to understand how this faulty result was computed, the developer would want to re-execute the program only with the input tuples that contribute to the faulty group.

Already in the first run, when the key of the faulty group is still unknown to the developer, automated instrumentation can be used to gather approximate provenance information (Figure 2B). The system attaches a provenance label p to each source tuple. The label is preserved during propagation through the data flow. Once the group key k for each tuple is computed, the system uses a set of probabilistic data structures, such as bloom filters, to track which provenance label contributed to which key[3]. In the second run of the program (2C), the developer knows the key of the faulty group (in the example we assume $K = b$). Consequently, the execution of the program can now be restricted to process only input tuples that belong to this key. To achieve this, we only consider tuples that pass the bloom filter that belongs to the faulty key. Depending on the selectivity of the bloom filter, this approach could significantly reduce the execution time.

Figure 2D presents sample tuple sets for the previously mentioned versions of the data flows. The unmodified datasource produces a set of values $v_1 ... v_8$. In the instrumented data flow for provenance tracing, these values receive a provenance label[4]. Based on their key k, we insert the provenance labels p of tuples into bloom filters. While we require each key to be assigned to exactly one bloom filter, multiple keys can share the same filter. In the second run, we now choose the bloom filter for the key of the faulty group $k = b$ and test it with all provenance labels of the data source. We can see that the filter returns true for at least all labels that belong to $k = b$. Additional labels may pass the filter, because we use one filter for several keys at once, and because of the probabilistic nature of bloom filters in general. Still, considerably less source tuples need to be processed to test the aggregation of all elements with $k = b$. After passing the bloom filter, we can project away the provenance label and perform the computation of k for each tuple. After this computation, we filter the remaining tuples to only contain entries with $k = b$. The aggregation operator now only receives tuples that contributed to the faulty aggregation result.

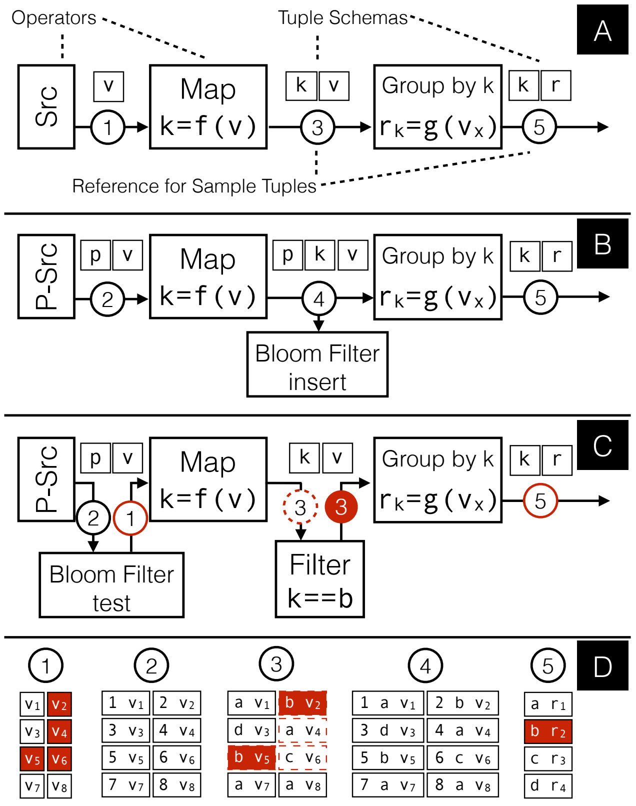

Figure 2: Illustration of Data Slicing. A: Original data flow. B: Instrumented data flow for provenance data collection. C: Instrumented data flow for efficient re-execution. D: Sample tuples for the different stages in the data flows in A, B, and C.

3. IMPLEMENTATION

The previously mentioned core techniques cannot reasonably exist on their own right, but require accompanying functionality that needs to be integrated into the execution

[3]Here, it is not necessary to have a one-to-one mapping between bloom filter and key, several keys could share a single bloom filter.

[4]While unique labels are used in the example, it should be noted that uniqueness is not a hard requirement for these labels

system. Among other things, the result of the instrumentation process will need to be communicated to the workstation of the developer and visualized. Doing this in a scalable way will require approaches like staged aggregation and efficient rendering. We intend to engage this issues in a best-effort basis using state-of-the-art strategies. If during the course of work we encounter potential for a completely new approach, we will follow it, however, without any clear indication for a novel solution, we will not pursue this actively. The Stratosphere system[5] will serve as basis for our prototypical implementations. However, in principle these concepts are transferable to other implementations like Spark[6] or Hadoop MapReduce[7].

It should be noted that our implementation focusses on the user-provided code that is executed on the system. Debugging the runtime environment itself is currently out of scope of this research. Consequently, we do not consider typical runtime issues such as race conditions or data partitioning and shuffling.

4. VALIDATION

The purpose of this section is to provide a clear definition of the goal we pursue with our research in order to determine whether the proposed techniques are fit for their purpose.

We argue that any given bug can be detected and analyzed manually by cyclic debugging, i.e. repeated execution and modification of the source of the program (*slicing* and *logging*) and subsequent log analysis [15]. The modification of the source would primarily serve to adjust the verbosity of the log and to enable/disable relevant code paths that are executed (in an attempt to minimize re-execution cost).

The *cost* of this process could therefore be modeled in terms of how many *code modifications* have to be performed to analyze (not fix) the bug and how long the *accumulated execution time*[8] of all runs is. We include the execution time of the run that is required to detect the bug in the first place.

The success of this work can be evaluated by comparing the costs of detecting and debugging a faulty program using the proposed techniques against the costs of detecting and debugging it with manual cyclic debugging strategies.

To simplify the comparison of both techniques, we make the following assumptions:

1. The production code runs without any default logging. Any logged information would need to be acquired through manual modification or automated instrumentation. Without this assumption, it would be impossible to assess the value of automated instrumentation.

2. Manual source modifications for analytical purposes are likely to introduce new bugs, which in turn must be analyzed manually. However, the cost of recursive bug introduction cannot be reliably quantified. We therefor control this variable by assuming that manual source modifications do not introduce new bugs. This assumption favors the manual approach and will be considered accordingly in the comparison.

3. AI, CUS, and DS are correctly implemented.

4. The cost of different code modifications cannot be compared against each other, we can only assume that no modifications cause less costs than any modification at all.

Given the previous assumptions, the three outlined techniques would achieve our goal if it was possible to analyze a given bug without any manual code modifications in less accumulated execution time.

5. RELATED WORK

The most similar approach to the one presented in here is the debugging suite that accompanies IBM's System S [6, 4]. IBM's approach provides means of tracking input and output tuples of nodes in stream processing operator graphs and supports local debugging of operators on data subsets. Compared to the ideas presented in this work, it lacks the instrumentation of UDFs and cross UDF code analysis.

Also very closely related to our approach is the *Lipstick* framework described by Amsterdamer et al. [1]. The system can generate provenance information of varying granularity from Pig Latin [17] workflows. The main points of differentiation with our work are that we seek to directly augment the user code as well the consideration of approximate provenance.

Perm a system described by Glavic and Alonso [7] seeks to provide provenance information of SQL query results. The core idea here is to rewrite a given SQL query in order to attach provenance information to the result tuples. Our concept of data slicing, which also enriches the user's code for gaining provenance information, is closely related to this approach, however in the context of Big Data, we propose to work with approximate provenance information that is not attached to each result tuple.

Crawl et al. [3] propose a provenance tracking system for Map/Reduce. Contrary to the concepts described here, Crawl et al. suggest that provenance information should be explicitly attached to records and retained throughout the entire data flow. While this is useful for archiving and bookkeeping purposes, it is a not justifiable overhead for debugging.

Grust et al. [8] present a tool for debugging SQL queries and sub expressions of these queries using actual data from the database. While our works shares the mindset of analyzing faulty query/program segments using actual query data, our work adapts more to the needs created by the presence of UDFs and takes into account the extreme size of the datasets.

Herschel et al. [9] describe a system for answering the question why certain tuples do not appear in the answer to a given SQL query. Following the idea of data provenance, the system traces back the hypothetical origin of such a *c-tuple* and provides an explanation what tuples would not to be added to the source relations to get the expected result. The approach is closely related to the ideas presented here, however, concentrates on relational systems and does not consider approximate provenance. On the other hand, our work models missing tuples simply as faulty results and does not explicitly trace hypothetical tuples back to their sources.

Jones et al. also strive to minimize debugging time, however, not in a parallel distributed system, but by parallelizing the search for bugs itself [13] by targeting multiple bugs.

[5]http://stratosphere.eu
[6]http://spark.apache.org
[7]http://hadoop.apache.org.
[8]Again, we take into account the CPU hours spent, not only wall clock time.

DUESTERWALD et al. [5] and MOHAPATRA et al. [16] present program slicing approaches that work across process boundaries, which communicate via message passing. Our take on Cross UDF Slicing extends this work into the realm of massively parallel data processing systems in which the different processes have better defined semantics.

HÖLZLE et al. [10] present means for debugging optimized code which has little resemblance to the original code written by the developer. This is an interesting topic, highly relevant to modern data processing systems. Currently, we do not consider the effects of dealing with optimized code. Assuming that UDF code is not restructured (only moved to different positions in the processing graph) and that the optimizer works correctly, mapping the UDFs to their new positions should be sufficient.

JIVE[18] and JOVE[19] are two subsequently developed systems that aim to visualize execution paths of running Java applications. This work is very relevant to the approach presented here, however, it needs to be adapted and extended to dealing with potentially thousands of parallel and remote processes.

A considerable amount of work has been done to debug multithreaded software based on MPI [14, 11]. Most of these approaches rely on logging the sequence of events in the different threads to ensure repeatability during the debugging stage. Deadlocks, for example, can be replayed and analyzed once they have been recorded. One could argue that these generalized approaches to debugging subsume the more specific ideas presented in this paper. However, we argue that recording the sequence of events in a massively parallel data processing system that operates on terabytes of data, is prohibitively expensive. We would also like to point out that the abstraction level of these approaches may be suitable for debugging the runtime engines of a massively parallel system, but not for a semantically enriched query in which defects like deadlocks are not an issue.

6. FUTURE WORK

We intend to research and prototypically implement the techniques outlined in Section 2 within the course of the following two to three years, aiming for completing this work within the year 2016.

Starting with Automated Instrumentation, we will familiarize ourselves with code analysis and code modification techniques and implement the data gathering infrastructure as well as some form of IDE support for selection of instrumentation points.

Building on top of the Automated Instrumentation infrastructure we will then research and implement the Data Slicing approach, which is likely to provide the most gains in terms of re-execution speed-up.

As a final step, we will focus on Cross UDF Slicing beginning with IDE based sanity checks and instrumentation point recommendation. This leaves the removal of non-relevant code as the last item. Here, we will start with low-hanging fruits like removal of complete operator subgraphs and work our way up to the deactivation of irrelevant UDF code paths.

7. ACKNOWLEDGMENTS

We would like to thank our advisor Prof. Volker Markl as well as Dr. Asterios Katsifodimos, Dr. Holmer Hemsen, Juan Soto, and Moritz Schubotz, as well as Prof. Stefan Edlich for their valuable feedback on this work. This work is partially funded by the European Commission under FP7 Project No. 296448 – "Data Supply Chains for Pools, Services and Analytics in Economics and Finance."

8. REFERENCES

[1] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *Proc. VLDB Endow.*, 5(4):346–357, Dec. 2011.

[2] P. Buneman, S. Khanna, and W. C. Tan. Why and Where: A Characterization of Data Provenance. *ICDT*, pages 316–330, 2001.

[3] D. Crawl, J. Wang, and I. Altintas. Provenance for mapreduce-based data-intensive workflows. In *Proceedings of the 6th Workshop on Workflows in Support of Large-scale Science*, WORKS '11, pages 21–30, New York, NY, USA, 2011. ACM.

[4] W. De Pauw, M. Leia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, and D. Sow. Visual debugging for stream processing applications. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Rou, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 18–35. Springer Berlin Heidelberg, 2010.

[5] E. Duesterwald, R. Gupta, and M. Soffa. Distributed slicing and partial re-execution for distributed programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 497–511. Springer Berlin Heidelberg, 1993.

[6] B. Gedik, H. Andrade, A. Frenkiel, W. De Pauw, M. Pfeifer, P. Allen, N. Cohen, and K.-L. Wu. Tools and strategies for debugging distributed stream processing applications. *Software: Practice and Experience*, 39(16):1347–1376, Nov. 2009.

[7] B. Glavic and G. Alonso. The perm provenance management system in action. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1055–1058, New York, NY, USA, 2009. ACM.

[8] T. Grust, F. Kliebhan, J. Rittinger, and T. Schreiber. True language-level SQL debugging. In *EDBT/ICDT '11: Proceedings of the 14th International Conference on Extending Database Technology.* ACM, Mar. 2011.

[9] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. In *Proceedings of the VLDB Endowment.* VLDB Endowment, Sept. 2010.

[10] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[11] C.-E. Hong, B.-S. Lee, G.-W. On, and D.-H. Chi. Replay for debugging MPI parallel programs. In *MPI Developer's Conference, 1996. Proceedings., Second*, pages 156–160. IEEE Comput. Soc. Press, 1996.

[12] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening

the black boxes in data flow optimization. In *Proceedings of the VLDB Endowment*. VLDB Endowment, July 2012.

[13] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 16–26, New York, NY, USA, 2007. ACM.

[14] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *Computers, IEEE Transactions on*, (4):471–482, 1987.

[15] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *Computing Surveys (CSUR*, 21(4):593–622, Dec. 1989.

[16] D. P. Mohapatra, R. Kumar, R. Mall, D. S. Kumar, and M. Bhasin. Distributed dynamic slicing of Java programs. *Journal of Systems and Software*, 79(12):1661–1678, Dec. 2006.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[18] S. P. Reiss. Visualizing java in action. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 57–ff, New York, NY, USA, 2003. ACM.

[19] S. P. Reiss and M. Renieris. Jove: Java as it happens. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 115–124, New York, NY, USA, 2005. ACM.

[20] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.