Automated analysis of algorithms implemented on top of TBD

Walter Tichy, Umut Acar, Emanuel Jöbstl
Karlsruhe Institute of Technology, Faculty of Computer Science
Carnegie Mellon University, Computer Science Department

Abstract—**The principle of incremental programming has been well known for many years. While the automatic transformation of sequential programs into efficient incremental programs is not solved in general, a number of platforms for incremental computation have been created. In this document, we inspect the incremental computation platform TBD. First, we summarize known principles like directed dependency graphs, execution traces, intrinsic trace distance and trace stability, then we create an program model which enables us to apply those principles to TBD programs. We also present an algorithm to calculate the intrinsic trace distance in practice. Finally, we show how we can make automatic optimizations to programs by analyzing the dependency graph.**

I. Introduction

This section is going to describe the purpose of incremental computation and also explain why the concept of incremental computation is useful for speeding up computations. Furthermore, used and referenced terminology should be outlined.

A. Approaches to incremental computation

This subsection briefly describes various approaches to incremental computation, and outline their strengths and weaknesses.

These approaches include:

- Providing a platform or framework for incremental programming, utilizing
  - function caching. [1] [2]
  - formal manipulation of the program. [3]
  - differential data flow. [4]
  - a combination of multiple approaches, like memorization and execution traces. [5] [6] [7] [8]
- Providing a high-level abstraction, like an incremental database. [9]
- Deriving an incremental program from a non-incremental, non-functional program. [10]
- Deriving an incremental program from a non-incremental, functional program. [11]

II. TBD

This section provides an outline of the TBD platform, a framework for incremental computation currently being developed at CMU. The framework is being developed in the Scala language, which enables us to exploit the reflection capabilities of Scala for analysis [12].

A. Programming interface

This subsection is going to describe the TBD core API. Since TBD relies on the use of patterns known from functional languages, these patterns should be described. Also, the constraints for and the responsibilities of the developer have to be specified. The section should be concluded with an example.

III. Theoretic fundamentals

This section should give a brief summarization of the principles described in the work of U. Acar et al. This is important, because we later have to adjust the definitions to fit the program model of TBD. [13]

A. Execution Traces

This subsection describes the approaches of using Traces and Directed Dependency Graphs (DDGs). [13]

B. Memorization

This subsection describes how traces and memorization together are used to accomplish incremental computing. [13]

C. Stable algorithms

This subsection describes the concepts of stable algorithms, intrinsic trace distance and their relationship. [1]

Also, this section should emphasis that the intrinsic trace distance forms a lower bound for the time needed by change propagation during an update. [13]

D. Program model for TBD

This subsection describes a program model for TBD, which will be used for theoretical conclusions in the following sections. The differences and similarities with the programming model in [13] should be highlighted.

IV. An Intrinsic trace distance algorithm for TBD

This section is going to describe our algorithm for calculating the minimal trace distance of execution traces produced by TBD. Also, implementation details and challenges during the implementation (like defining equality for anonymous functions with free variables in practice) should be discussed.

[1]Intrinsic trace distance is a central concept for this work and can basically be described as an edit distance between two trees. The definition can be found in [13], chapter 7 or [14].

A. *Proof of correctness*

We shall also proof the correctness of our algorithm. It is of importance that we take the changes of the program model outlined in the previous section into account.

V. Automatic optimization of programs

This section is going to describe how analyzing the Directed Dependency Graph (DDG) can be used for automatic optimization. For accomplishing this task we can utilize the following features of the DDG:

- Caller/callee dependencies.
- Dependencies of modifiables[2].
- Dependencies of bound variables which are not modifiables.

Furthermore, using the intrinsic distance algorithm, we can recognize which nodes are deleted, inserted or retained [13], which can be used to optimize the program to accomplish faster change propagation.

The exact contents described in this chapter are still to be determined, based on our findings. Possible approaches include, but may not be limited to:

- Function call reordering.
- Insertion of explicit memorization calls.
- Detection of cascading updates, which could be omitted.

VI. Evaluation

This section is going to demonstrate the usefulness of the described techniques using real-world algorithms, like map, reduce and quicksort.

Basically, it is shown how it is possible to optimize a classic implementation (without memorization) of each algorithm, so that change propagation time lies within the same complexity class as the theoretical lower bound for updates for this algorithm.

VII. Conclusion

This section will conclude and summarize with the findings of this work.

A. *Future work*

The final section briefly outlines problems encountered but not solved during the writing of this thesis, as well as encourages future research on interesting issues of incremental computation.

[2]Pointer-like variables which have to be explicitly read and written, and therefore support automatic change propagation

References

[1] A. Heydon, R. Levin, and Y. Yu, "Caching function calls using precise dependencies," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 311–320, 2000.

[2] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1989, p. 315328. [Online]. Available: http://dl.acm.org/citation.cfm?id=75305

[3] R. F. Cohen and R. Tamassia, "Dynamic expression trees and their applications," in *SODA*, 1991, pp. 52–61.

[4] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray, "Composable incremental and iterative data-parallel computation with naiad," Tech. Rep. MSR-TR-2012-105, October 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=174076

[5] M. A. Hammer, U. A. Acar, and Y. Chen, "CEAL: a C-based language for self-adjusting computation," in *ACM Sigplan Notices*, vol. 44. ACM, 2009, p. 2537. [Online]. Available: http://dl.acm.org/citation.cfm?id=1542480

[6] Y. Chen, U. A. Acar, and K. Tangwongsan, "Functional Programming for Dynamic and Large Data with Self-Adjusting Computation," 2014. [Online]. Available: http://www.mpi-sws.org/~chenyan/papers/icfp14.pdf

[7] U. A. Acar, A. Ahmed, and M. Blume, "Imperative self-adjusting computation," in *ACM SIGPLAN Notices*, vol. 43. ACM, 2008, p. 309322. [Online]. Available: http://dl.acm.org/citation.cfm?id=1328476

[8] U. A. Acar, G. E. Blelloch, and R. Harper, "Adaptive functional programming," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 6, pp. 990–1034, 2006.

[9] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications." in *OSDI*, vol. 10, 2010, p. 115. [Online]. Available: https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Peng.pdf?origin=publication_detail

[10] Y. A. Liu and T. Teitelbaum, "Systematic derivation of incremental programs," *Science of Computer Programming*, vol. 24, no. 1, pp. 1–39, 1995.

[11] R. Ley-Wild, M. Fluet, and U. A. Acar, "Compiling self-adjusting programs with continuations," in *ACM Sigplan Notices*, vol. 43, no. 9. ACM, 2008, pp. 321–334.

[12] E. Burmako, "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming," in *Proceedings of the 4th Workshop on Scala*. ACM, 2013, p. 3.

[13] U. A. Acar, "Self-adjusting computation," 2005.

[14] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo, "Dynamizing static algorithms, with applications to dynamic trees and history independence," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2004, pp. 531–540.