Language Support for Efficient Dynamic Computation

Umut A. Acar
Carnegie Mellon University
umut@cs.cmu.edu

Ezgi Cicek
MPI-SWS
ecicek@mpi-sws.org

Deepak Garg
MPI-SWS
dg@mpi-sws.org

Abstract

While programming language researchers have the techniques and tools to design and develop programming languages that can improve the algorithmic—not just runtime—efficiency of computations, they usually avoid doing so, leaving this problem instead to algorithm designers. This approach results in a clean division of labor: programming language researchers focus on correctness and semantics and the algorithms researchers focus on algorithmic efficiency. It can, however, lead to undesirable outcomes such as impractical, overly complicated algorithms, and inefficient programming languages. We feel that integrating concerns of algorithm efficiency into the language design can improve the impact of programming languages research. In this paper, we propose to study a notion of continuity which we call dynamic optimality, for the purpose of improved efficiency of computations operating on dynamically changing data.

1. Introduction

Many real world applications operate on data that changes dynamically over time. For example, internet or social-network graphs, which represent the connections between web sites or "people" are continuously changing, dynamic structures. To remain relevant and up to date, applications that operate on such data must allow changing the data and must be able to respond to such changes quickly and efficiently. Such responsiveness and efficiency is especially important because data sizes have been increasing (exponentially by some measures), while computational power has essentially stagnated.

This convergence of events has increased the interest on incremental or dynamic computation, where application can respond to dynamically changing data. The idea behind incremental computation is to structure the computation of an output from an input in such a way that, when the input changes slightly, sub-computations can be re-used to produce and update the output efficiently, without having to re-perform the whole computation with the new input.

The problem of how to develop such incremental/dynamic applications is an old problem, going back to early 80's, and has been studied independently in algorithms and programming languages communities. In the algorithms community, researchers developed *dynamic algorithms* which are carefully designed to achieve efficiency under small updates to their input. This vast research area (e.g., [5]) shows that dynamic algorithms can be asymptotically more efficient than their conventional counterparts. Dynamic algorithms can, however, be difficult to develop and implement even for problems that are simple in the static case where data is not allowed to change. In the programming languages community, research developed to enable automatic incrementalization for a broad range of computations. The large body of research on incremental computation showed that it can be possible to achieve generality and efficiency (e.g. [8]) in some specific instances. Recent advances on self-adjusting computation developed techniques that can achieve efficiency and full generality (e.g., [1, 2, 7]). Techniques for parallel self-adjusting computation have also been proposed [3, 6].

While previous research has advanced the state of the art on computing with dynamically changing data, all previous approaches require designing algorithms and software with efficiency in mind. In algorithmic approaches, efficiency concerns are entirely explicit; in fact, this is often the whole point of algorithmic research. It is somewhat less explicit, but still very important in self-adjusting computation and other language-based approaches: the software designer still must reason about efficiency, though at a somewhat higher level. We are interested in studying the design of high-level programming languages that can guarantee efficiency for dynamically changing data sets. Our goal is to design expressive programming languages where computations can respond to change to their data automatically and efficiently performing work or taking time proportional to the size of the change in the data, i.e., the "delta", rather than the size of the whole data. Tackling this challenge appears to require using a methodology that combines techniques and tools from programming languages and algorithms, which we believe to be interesting, as it brings together two main areas of theoretical computer science.

The key question is how to formulate such a general theory. Our proposal, which we outline in the next section, is inspired by insights from much work in the algorithms and programming languages community. Essentially all approaches to the problem of dynamic or incremental com-

putation (the problem of responding efficiently to changing data) are based on the idea of structuring the computation in such a way that when the input changes slightly the set of sub-computations that must be performed to compute the new result do not change substantially. In other words, they ensure that computation remain continuous: small perturbations lead to small changes in the computation [4]. Our approach, *dynamic optimality*, builds on this idea and formalizes *optimally efficient propagation of input changes* to output.

2. Dynamic Optimality

We explain our proposed technical approach and define dynamic optimality abstractly. To keep the exposition simple and clean, we omit constant factors from definitions and examples. We start by illustrating dynamic optimality on an example in the nested relational calculus (NRC), an abstract language of database programs. The motivation for working with NRC comes from applications on databases and big-data and, in particular, map-reduce programs.

Consider the NRC expression $R = \texttt{map}\{f(x) \mid x \in S\}$ on some set S. This expression simply returns another set by mapping a function f over the members of S. Suppose we compute R for some set S and then wish to recompute R for a new set $S_1 = S \cup \{a\}$ that adds a new element a to S. The question is: How efficiently can we do this, given the entire computation at S? It is easy to see that we only need to compute afresh $f(a)$ and re-use most of the earlier computation. Hence, the time taken to re-compute R at S_1 is the time needed to compute $f(a)$ (plus some constant time to reconstitute the output set, which we ignore here). The next question is: Can we argue that this time is optimal? Again, it is easy to see that, asymptotically, this is the best we could do: If, instead, there were a general algorithm to update the output after inserting element a that ran in time less than the time needed to compute $f(a)$, then starting from an empty set and adding elements of S one by one, we could compute R on S in less time than needed to compute $f(s)$ for each $s \in S$, which is absurd. Hence, performing *optimal* incremental computation for the R is easy. Our notion of dynamic optimality (described next) formalizes this optimality of incremental computation for programs in general.

Fix a programming language L and its operational semantics, which, for each terminating expression e in the language, defines the derivation tree $\mathcal{D}(e)$ of the finite computation of e. To define dynamic optimality, assume a metric space on expressions — $\Delta(e, e')$ — that measures the *distance* between expressions e and e'. Using the same notation, let $\Delta(\mathcal{D}(e), \mathcal{D}(e'))$ denote the distance between the computations of e and e'. $\Delta(\mathcal{D}(e), \mathcal{D}(e'))$ is a measure of the time needed to produce the output of e', given an earlier computation of e.

Let e be a program and e_1, e_2, e_3 denote three arbitrary variants of it. We say that e is dynamically optimal if $\Delta(e_1, e_2)+\Delta(e_2, e_3) = \Delta(e_1, e_3)$ *implies* $\Delta(\mathcal{D}(e_1), \mathcal{D}(e_2))+\Delta(\mathcal{D}(e_2), \mathcal{D}(e_3)) = \Delta(\mathcal{D}(e_1), \mathcal{D}(e_3))$. In other words, if we take a *monotonic* sequence e_1, e_2, e_3 of changes to e, formalized by $\Delta(e_1, e_2) + \Delta(e_2, e_3) = \Delta(e_1, e_3)$, then the time needed to update output from the computation of e_1 to e_3 is the sum of the time needed to update from e_1 to e_2 and then from e_2 to e_3.

Why does this definition capture optimality of dynamic update? By definition of a metric, it is always the case that $\Delta(\mathcal{D}(e_1), \mathcal{D}(e_2))+\Delta(\mathcal{D}(e_2), \mathcal{D}(e_3)) \geq \Delta(\mathcal{D}(e_1), \mathcal{D}(e_3))$. By forcing equality, we force that both $\Delta(\mathcal{D}(e_1), \mathcal{D}(e_2))$ and $\Delta(\mathcal{D}(e_2), \mathcal{D}(e_3))$ be as minimal as possible. Choosing $\Delta(S, S')$ to be the size of symmetric set difference of S and S' in our earlier example, it is easy to see that this idea mirrors our earlier optimality argument.

This definition leads to many interesting questions, e.g., how do we check dynamic optimality, is there a language in which all programs are optimally updatable, what metrics to use, and what metrics on inputs coincide with practice? In future work, we plan to look at these questions. In particular, we intend to carry out an extensive study of database languages like NRC.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6): 990–1034, 2006.

[2] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32 (1):3:1–53, 2009.

[3] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

[4] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity and robustness of programs. *Communications of the ACM*, 55(8):107–115, 2012.

[5] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005.

[6] Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

[7] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[8] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages*, pages 502–510, 1993.