Self-Adjusting Computation
(An Overview)

Umut A. Acar
Toyota Technological Institute at Chicago
umut@tti-c.org

Abstract

Many applications need to respond to incremental modifications to data. Being incremental, such modification often require incremental modifications to the output, making it possible to respond to them asymptotically faster than recomputing from scratch. In many cases, taking advantage of incrementality therefore dramatically improves performance, especially as the input size increases. As a frame of reference, note that in parallel computing speedups are bounded by the number of processors, often a (small) constant.

Designing and developing applications that respond to incremental modifications, however, is challenging: it often involves developing highly specific, complex algorithms. Self-adjusting computation offers a linguistic approach to this problem. In self-adjusting computation, programs respond automatically and efficiently to modifications to their data by tracking the dynamic data dependences of the computation and incrementally updating their output as needed. In this invited talk, I present an overview of self-adjusting computation and briefly discuss the progress in developing the approach and some recent advances.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design, Algorithms, Performance, Experimentation.

Keywords Incremental Computation, Self-adjusting Computation, Compilers, Run-Time Systems, Memory Management, Computational Geometry, Scientific Computing, Machine Learning, Computational Biology.

1. Motivation

Since the early days of computer science, researchers realized that many uses of computer applications are *incremental* by nature. We often start with some initial input and obtain some initial output. We then repeatedly observe the output, make some small modifications to the input, and re-compute the output. In many cases, such incremental modifications cause only small modifications to the output making it feasible to update the output asymptotically more efficiently than recomputing from scratch when the input is modified. In some cases, this asymptotic improvement is crucial to making computations tractable in practice.

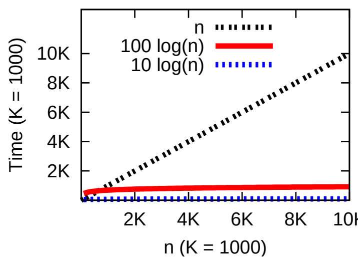

Figure 1. Linear time versus logarithmic time (linear scale).

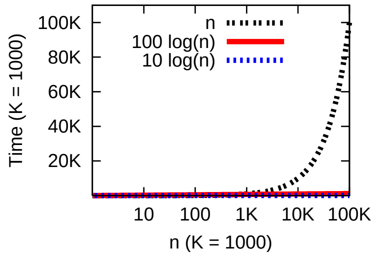

Figure 2. Linear time versus logarithmic time (logarithmic scale).

Examples of incrementality abound. In some applications, input modifications arise as a result of interaction with external agents. For example, applications that interact with or model the physical world (e.g., robots, traffic control systems, scheduling systems) must respond to modifications in the world as it evolves. Similarly applications that interact with the user must respond to his/her modifications, e.g., in software development, the compiler is invoked repeatedly after the user makes small modifications to the program code. In other examples, input modifications can be inherent to the application. For example, in motion simulation, objects move continuously over time causing the property being computed to change continuously as well. For example, we can simulate the flow of a fluid by modeling its constituent particles and computing properties of the moving particles (e.g., their triangulation) while updating them as points move. Since motion often causes small combinatorial modifications to computed properties, we can often model it as a sequence of incremental modifications.

I would like to start by going through a "back-of-the-envelope" calculation to illustrate the potential improvements offered by incremental changes. Let's assume that an interesting application takes at least linear time in the size of the input[1] and that we can have it respond to incremental modifications in logarithmic time by performing an incremental update. Figure 1 shows how the times for re-computing from scratch (linear) and updating (logarithmic time) grow with the input size. Since the logarithmic update time is reasonably expected to have a larger constant factor, we consider the constant factors 10 and 100. Note that re-computing from-scratch dominates incremental updates starting at small input sizes (in 100's) and the speedup obtained by incremental updates increases linearly with the input size. For example, when the input exceeds three orders of magnitude so does the speedup. As a frame of reference compare this to parallel computing, were speedups are bounded by the number of processor, often a (small) constant. Note also that re-computing from-scratch is exponentially slower than updating incrementally, i.e., $n = 2^{\log n}$ (Figure 2). Thus if we can achieve (poly-) logarithmic time incremental update times, then we can hugely speedup computations. This can also make realistic some applications that are otherwise impractical.

Having realized the potential offered by incrementality, previous work devised techniques for enabling computations respond efficiently to incremental modifications (e.g., see [24, 15, 14] for surveys). Except in some special cases, this requires designing specific algorithms or data structures for remembering and re-using results so that computed properties of interest may be updated efficiently. Until recently no broadly effective general-purpose technique or language was known for developing applications that can interact with changing data. Self-adjusting computation proposes a solution to this problem.

2. Self-Adjusting Computation

The goal of self-adjusting computation is to enable the development of software that can take advantage of incremental updates. The approach is language centric: instead of expecting the programmer to develop software for a particular application and a particular class of incremental modifications, we offer a language-based approach where any program can automatically respond to incremental modifications.

In self-adjusting computation, programs consist of two components: a self-adjusting core and a top- or meta-level mutator. The self-adjusting core performs a single run of the intended application with fixed, unchanging input data. The mutator drives the self-adjusting core by supplying the initial input and by subsequently modifying the input or other computation data. The mutator can modify computation data in a variety of forms depending on the application. For example, it can insert a new object into a set of objects that make up the input to a self-adjusting program, or it can change the outcome of a comparison performed during the execution. After performing such modifications, the mutator can update the output and the computation by requesting *change propagation* to be performed. Change propagation automatically updates the computation and the output by propagating the data modifications through the core.

By supplying an automatic change-propagation mechanism, self-adjusting computation shifts the burden of designing and implementing an incremental-update mechanism from the programmer to the language designer and implementor. The language designer in turns needs to answer a number of questions:

1. Can we design a general-purpose mechanism that can guarantee efficient response times under a broad range application domains?
2. What should be the primitives of the language for writing self-adjusting programs?
3. Can we compile self-adjusting programs efficiently?
4. Can we provide a cost model for reasoning about the asymptotic complexity of self-adjusting programs?
5. Can we extend existing languages to support self-adjusting computation?
6. Can the approach be effective in practice, e.g., compared to ad hoc approaches?

I give an overview of the advances that my collaborators and I have made in answering these questions and point out some of the remaining challenges.

2.1 DDGs, Memoization, and Change Propagation

To update computations efficiently, we represent executions of programs with *dynamic dependence graphs* or *DDGs* for short. At a high-level, DDGs record all changing data, input and intermediate, in a computation and the dependences between the data and the code that use the data. Given the DDG of an execution of a program P with some inputs I and modified input I', change propagation updates the DDG as if P is executed from-scratch with the modified input I'. To achieve this, change propagation finds and re-executes the code that depend on the modified data. Since re-executing code can modify other data, change propagation continues until all modifications are processed.

DDGs and change propagation were proposed in a paper in 2002 [5]. The same paper proposed language facilities for writing so called adaptive programs that can automatically respond to modifications via change propagation. The key idea behind these facilities is the notion of *modifiable references* which hold data that changes over time. The language facilities enable tracking dependences selectively by providing a particular interface to modifiables that makes explicit the dependences between modifiable data and the code. More specifically, the mutator can update only the contents of modifiables, and the contents of modifiables can be accessed only within the scope of a special *read* primitive that can "return" a value only by writing to another modifiable. With these restrictions, change propagation can be performed by recording in the DDG only the operations on modifiable references. This selective tracking of dependences is critical to ensure that the approach can be supported efficiently in practice.

A major limitation of DDGs is that change propagation can re-execute code unnecessarily, because re-executed code cannot re-use previously performed computations. Subsequent work [4, 10] remedy this limitation by integrating memoization with DDGs and change propagation by showing that they are essentially duals that, when combined, provide for efficient re-use. Due to a number of conflicting requirements between memoization and DDGs, (e.g., memoization requires purely functional code whereas propagation imperatively updates memory), this integration is conceptually complex. We propose a particular technique for integrating them and prove it correct by providing mechanically checked proofs [10]. We prove that the proposed approach is also efficient [4]. The term "self-adjusting computation" was first used to refer to computations that use this approach to update computations.

2.2 Effects

Initial work on self-adjusting computation assumed a purely functional setting: the core would not be allowed to side-effect the memory or perform other effectful computations such as writing to a file. The difficulty with effects is that change propagation can skip them because it only re-executes parts of the computation—thus, the se-

[1] This assumption is justified because an interesting application should at least inspect all of its input (thus requiring linear time).

mantics differs from that of a from-scratch run. This do not seem to be a problem for certain benign effects such as writing to the screen, but not for other effects such as writing to a file or updating memory.

Follow up work generalized self-adjusting computation primitives to support modifiable references that can be updated imperatively in the core [3]. (Note that the mutator is always allowed to update modifiables destructively.) The idea is to record different versions (contents) that a modifiable may have and track dependences at the granularity of versions instead of modifiables themselves. We proposed techniques for performing dependence tracking and change propagation in the presence of memory effects but there appears to be room for asymptotic improvements (currently they can cause a logarithmic time slowdown). Apart from memory effects, the interaction between self-adjusting computation and other effects such as I/O is not well understood.

2.3 Language primitives

Although self-adjusting computation can be applied without having to change existing code by tracking all data and all dependences between code and data [2], this is prohibitively expensive in practice. Early work on self-adjusting computation therefore proposed primitives for operating on modifiable references and for memoizing functions to enable tracking dependences selectively [5, 4]. These primitives effectively require the programmer to mark explicitly the dependences between code and data.

These proposals suffer from several major limitations: 1) they can require significantly restructuring existing code, 2) they can make reasoning about performance and complexity difficult (there are many ways to write a program with significantly different performance), 3) they are higher-order, which makes them unsuitable for lower level languages such as C or Java. In the past several years, we therefore have been developing an alternative, compilation-based approach that eliminates the difference between self-adjusting computation primitives and the primitives for conventional imperative languages [21, 18]. The idea is to allow the programmer to operate on modifiable references just like conventional references by reading/dereferencing and writing them without restrictions. The approach shifts the burden of finding the dependences between code and data from the programmer to the compiler.

2.4 Compilation

To compile a conventional (imperative) program into a self-adjusting program, we need to find the part of the code that depends on each use of a (modifiable) references contents. One idea is to use the continuation[2] of each dereference operation as a conservative approximation [21, 20]. Continuations, however, approximate actual dependences coarsely: a small modification can require change propagation to re-execute a continuation completely, which extends to the end of the computation. We therefore memoize continuations themselves and devise a technique for memoizing functions with continuation arguments such that they can be re-used even if their continuation arguments do not match. We make this intuitive description precise by formulating a program translation, based on a continuation-passing-style (*cps*) transform, for compiling conventional imperative programs with first-class mutable references into self-adjusting programs [21, 20]. It is likely that the proposed translation is optimal for a reasonably large class of computations, i.e., it enables re-use of computation as much as possible, but we have no proof of this.

[2] Intuitively, a continuation at a point in the computation is a closed function that represents the rest of the computation.

Continuations and cps-translation come naturally in languages such as ML or Haskell, but they are cumbersome in imperative languages, e.g., in C. We have therefore been developing static analyses for identifying dependences between code and data directly from program code [18]. At a high level, the idea is to delimit the part of the program code that depends on each access to a modifiable and functionalize it by creating a function from the delimited code. Change propagation uses these functions to update the computation when the contents of a modifiable changes. Since this code transformation re-organizes the code by altering the control flow, care must be taken to ensure that the translation does not change the semantics of the program itself.

2.5 Implementations

There are two major ongoing efforts in realizing self-adjusting computation by extending the SML and the C languages. These extended languages called SaSML and CEAL (respectively) use the described compilation techniques (based on continuations or static analysis respectively) to compile conventional programs written in a natural style into self-adjusting programs. We have implemented (prototype) compilers for these languages. For SaSML [21], we extended the MLton compiler [1]. For CEAL [18], we developed a compiler by using the CIL framework [23]. The CEAL language performs very efficient automatic garbage collection by integrating garbage collection and change propagation [17]. These implementations can incur moderate overheads (in SaSML, overhead sometimes exceeds an order of magnitude, in CEAL it is less than a factor of five) when re-executing from scratch but can respond to changes asymptotically faster (often by a linear factor) than recomputing from scratch. In practice, we measure orders of magnitude speedups (we measured up to 6 orders).

2.6 Cost Model

Since self-adjusting programs are written like conventional programs but can respond to data modifications automatically via change propagation, their run-time behavior is quite different than their conventional interpretation would imply. It is therefore critical to provide a cost model that makes it possible for the programmer to reason about the efficiency of self-adjusting programs under incremental modifications. Before the development of compilation techniques, this was difficult because the previously proposed language facilities could obscure the run-time behavior. But now this goal is within reach. In recent work, we develop a notion of derivation or trace distance for analyzing the asymptotic complexity of self-adjusting programs under change propagation [20]. The idea is to compare the derivations for from-scratch runs of the program with differing inputs and compute the distance between them. We then show that change propagation takes time proportional to this distance.

An interesting aspect of the proposed cost semantics is that it allows reasoning about asymptotic complexity by making it possible to generalize and compose distances—since language-based cost models traditionally consider concrete evaluations, this is generally difficult. This work is still a first step. The cost model is nondeterministic—there are many distances between two derivations—and the implementation (the run-time system) does not necessarily choose the smallest distance.

2.7 Applications

To evaluate the effectiveness of self-adjusting computation, we have been investigating its applications to several problem domains. I review some of our work in the areas of computational geometry & scientific computing, and machine learning & computational biology.

Many scientific computing applications such as physical simulations requiring computing various geometric properties of changing data, e.g., a blood flow simulation requires computing various forces between molecules as they flow through the veins. In such applications, it is critical, for efficiency reasons, to update incrementally the computed properties. Unfortunately, these applications often require sophisticated algorithms making ad hoc incrementalization difficult. Thus many of these problems remain unsolved from a theoretical perspective and/or pose significant implementation challenges. Motivated by these examples, we have considered a number of fundamental computational-geometry problems such as computation of convex hulls and triangulations in both 2 and 3 dimensions. Our work [4, 8, 19, 7, 11] show that self-adjusting programs for various computational geometry problems respond to incremental modifications due to motion or external actions by a near linear factor faster than recomputing from scratch, yielding orders of magnitude speedups in practice. Using our approach we have also made progress on some difficult problems such as motion simulation of three-dimensional convex hulls [7] that has remained open for nearly a decade.

As another major application area we have been investigating the problem of incrementally updating statistical or Bayesian inferences under modifications to the assumed models. Statistical inference is a fundamental problem in machine learning that has broad applications. Our original motivation was to facilitate modifying the structure of proteins or other molecules to help enhance scientific understanding of their crucial functions. We show that statistical inferences can be updated under incremental modifications to underlying models in logarithmic time (as opposed to the linear time required by a from-scratch run) in the number of nodes [12, 13]. This problem has been previously formulated but remained open except for simple modifications that do not change the structure of the model. Our experiments with actual protein data show that these approaches can speedup the computation of various biological properties of proteins (total energy, three-d confirmation) by as much as an order of magnitude. Some of this work is based on our earlier work [6, 9] that showed that self-adjusting version of an algorithm for computing properties of trees [22] is competitive with optimal algorithms for incremental updating of tree computations [25].

3. Status and Future Work

In the past five years, we have developed some of the foundations for self-adjusting computation and investigated its practical effectiveness by extending existing languages and considering some applications. We can now compile (imperative or purely functional) programs written in a conventional, direct style to self-adjusting programs. Our compilers produce code that incurs moderate overheads when running from scratch but can respond to incremental updates (the common case) very quickly often by orders of magnitude faster than recomputing from scratch. For some of these application, we can also prove that self-adjusting computation closely matches best known or optimal bounds, even for problems that resisted ad hoc approaches. In the future, we plan to explore a number of problems, which I briefly mention below.

Language Design. We need a more expressive, type safe meta language. The current meta language is not, e.g., it allows the mutator to reference data that may become invalid, or garbage, during change propagation. In addition, the language does not allow self-adjusting computations to be treated as first class values or establish interesting connections between them, other than treating them as one large computation.

We need a better understanding of input/output (I/O) and how to support it efficiently. Existing languages do not allow I/O, though in practice, printing to the screen seems to "work."

Currently, our languages require programmer guidance to ensure efficient change propagation: the programmer must provide reasonably fine-grained hints indicating how memory locations should be re-used during change propagation, because the choice of locations affect computation re-use. We wish to reduce the need for programmer provided hints by expanding the run-time system and the language.

Language Implementation. Our existing implementations target a high-level language, SML, and a low-level language C. The C implementation is particularly efficient. But both implementations leave a lot of room for further optimizations—except for a few basic optimizations they support none. One particularly interesting optimization is to allow the run-time system to control the granularity of dependence tracking so that we can trade time for space.

Self-adjusting computation makes it critical to support automatic memory management, because during change propagation allocated objects may become unreachable/garbage making explicit memory management difficult. Our C implementation provides for efficient garbage collection by integrating it with change propagation. The technique, however, is conservative: it can treat memory objects live even if they are not. The SML implementations use the memory management facilities of their host compiler (SML/NJ or MLton). Since self-adjusting computation violates some critical assumptions made by conventional approaches to garbage collection, with both compilers we observe that performance degrades significantly as the resident-size of the computation approaches the size of the physical memory.

We need efficient techniques to support mutable references, which, in the current design, can require non-constant overheads, and an evaluation of the effectiveness of the approach with highly imperative programs.

Algoritmic Aspects and Cost Semantics. Our recent work shows that self-adjusting computation can yield optimal update times for a range of applications even for problems that resist ad hoc approaches. These results indicate that it may be possible to study incremental problems more systematically. They are, however, somewhat indirect: they often analyze self-adjusting programs by treating them as algorithms integrated with change propagation. Our recent proposal on cost semantics alleviates some of these concerns by facilitating source-level reasoning. It does not, however, provide a provably efficient implementation that can guarantee optimal response times. The approach suggests that it may be possible to provide a fully formal cost semantics for algorithmic analysis, e.g., making it possible for mechanized theorem provers or other mechanized tools to aid in analyzing the complexity of self-adjusting programs.

Parallelism. Parallel programs tend to be amenable to self-adjusting computation (in ways that are not fully understood) suggesting that it may be possible to reap the benefits of both. Except for some preliminary results [16], it is not known how self-adjusting computation may be made to work in the parallel setting. Also, it might be possible for self-adjusting computation to aid in parallelization of effectful programs by allowing change propagation to update computations instead of aborting due to side effects.

References

[1] MLton. http://mlton.org/.

[2] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[3] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2008.

[4] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[5] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive Functional Programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[6] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[7] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*, September 2008.

[8] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic Algorithms via Self-Adjusting Computation. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pages 636–647, September 2006.

[9] Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2005.

[10] Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. In *Proceedings of the 16th Annual European Symposium on Programming (ESOP)*, 2007.

[11] Umut A. Acar, Benoit Hudson, Kanat Tangwongsan, and Duru Türkoğlu. Maintaining well-spaced point sets under dynamic changes, November 2008. In preparation.

[12] Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian Inference. In *Neural Information Processing Systems (NIPS)*, 2007.

[13] Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Inference on General Graphical Models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008.

[14] Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

[15] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

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

[17] Matthew A. Hammer and Umut A. Acar. Memory management for self-adjusting computation. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 51–60, 2008.

[18] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: A C-based language for self-adjusting computation. Technical report, Toyota Technological Institute, November 2008.

[19] Benoît Hudson. *Dynamic Mesh Refinement*. PhD thesis, Carnegie Mellon University Computer Science Department, December 2007.

[20] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2009.

[21] Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2008.

[22] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 487–489, 1985.

[23] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.

[24] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 502–510, 1993.

[25] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.