

An Experimental Analysis of Self-Adjusting Computation

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

Guy E. Blelloch
Carnegie Mellon University
blelloch@cs.cmu.edu

Matthias Blume
Toyota Technological Institute
blume@tti-c.org

Kanat Tangwongsan
Carnegie Mellon University
ktangwon@cs.cmu.edu

Abstract

Dependence graphs and memoization can be used to efficiently update the output of a program as the input changes dynamically. Recent work has studied techniques for combining these approaches to effectively dynamize a wide range of applications. Toward this end various theoretical results were given. In this paper we describe the implementation of a library based on these ideas, and present experimental results on the efficiency of this library on a variety of applications. The results of the experiments indicate that the approach is effective in practice, often requiring orders of magnitude less time than recomputing the output from scratch. We believe this is the first experimental evidence that incremental computation of any type is effective in practice for a reasonably broad set of applications.

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

General Terms Languages, Performance, Algorithms.

Keywords Self-adjusting computation, memoization, dynamic dependence graphs, dynamic algorithms, computational geometry, performance.

1. Introduction

In many applications it can be important or even necessary to efficiently update the output of a computation as the input changes, without having to recompute it from scratch. The algorithms community has made significant progress on data structures that efficiently maintain the output for a specific problem as the input is updated (see *e.g.*, [16, 19] for surveys). These are referred to as *dynamic algorithms* or *data structures*. Although there have been hundreds of papers on this topic, and important theoretical advances have been made, few of the algorithms have been implemented—the algorithms can be quite complex, and it is hard to compose the algorithms to make larger components.

Over the same period of time the programming language community has made significant progress on run-time and compile-time approaches that dynamize standard static algorithms. These approaches are often referred to as *incremental computation*, and the idea is to maintain certain information during an execution of the static algorithm that can be used to efficiently update the output when the input is changed. When incremental computation can be made to work, it has at least two important advantages over dynamic algorithms—there is no need to program, debug, document, maintain, and augment separate static and dynamic algorithms for every problem, and composability of dynamic versions follows from composability of their static counterparts.

Important early work on incremental computation included the work of Demers, Reps and Teitelbaum [17] on *static dependence graphs*, and Pugh and Teitelbaum [30] on *function caching* (also called memoization). Recent work on *dynamic dependence graphs* [4, 13], and combining these with memoization (called *self-adjusting computation* [2, 3]) has significantly extended the applicability of the approach. In several cases it has been shown that the algorithms that result from dynamizing a static algorithm are as efficient as the optimal dynamic algorithm for the problem [6, 8, 2]. The results that have been presented, however, have mostly been theoretical—they specify asymptotic bounds on update times for a variety of problems, and prove correctness of the transformations, but do not analyze the performance of the approach in practice. Since the approach involves maintaining a complete dependence graph and memoizing many of the function calls, one indeed might wonder about the efficiency in practice.

This paper describes an implementation of the approach and presents experimental performance results. The implementation is based on a library written in Standard ML (SML). The library presents a simple interface for a user to instrument static code to use the approach—this interface is described elsewhere [3].

Section 3 presents the main algorithms and data structures used in the implementation of the library. These algorithms combine dynamic dependence graphs and memoization to effectively adjust computations to changes. Due to interactions between dynamic dependence graphs and memoization, the combination maintains some critical invariants and properties to ensure correctness and efficiency. A key part of the section is a description of how time-intervals can be used to maintain these properties and invariants.

Section 4 describes the implementation of the algorithms along with key features that were important in achieving good performance, including maintaining a space integrity invariant by eagerly deleting items from memo tables and the dependence graph struc-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006...\$5.00.

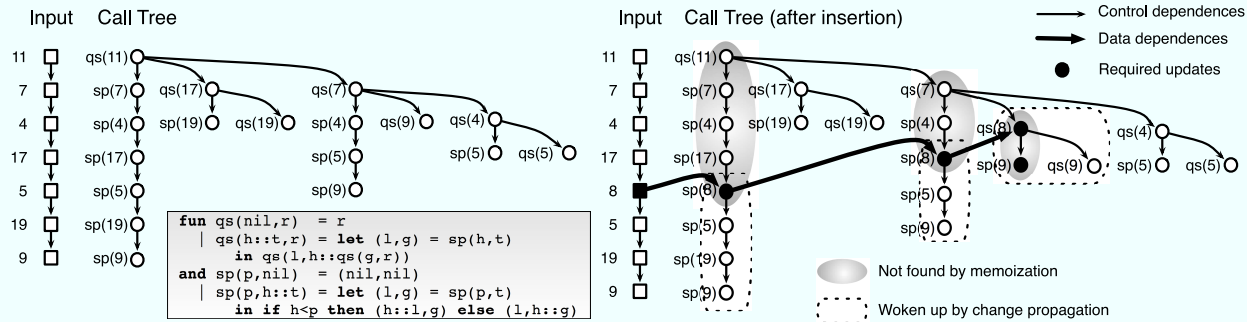


Figure 1. Memoization versus change propagation for quick-sort (qs). The numbers in parentheses are the first element of the argument list for the corresponding function call.

ture. An earlier implementation without this feature suffered from serious memory problems.

Section 5 describes a number of applications including a number of list primitives (filter, fold, map, reverse, split), two sorting algorithms (merge-sort, quick-sort), and some more involved algorithms in computational geometry including a number of convex hull algorithms (Graham-scan [20], quick-hull [9], merge-hull, Chan’s ultimate convex hull [14]), and an algorithm for maintaining the diameter of a point set [33]. These applications are chosen to span a number of computing paradigms including simple iteration (filter, map, split), accumulator passing (reverse, quick-sort), incremental result construction (Graham-scan, diameter), random sampling (fold), and divide-and-conquer (quick-sort, merge-sort, quick-hull, merge-hull). Section 6 evaluates the effectiveness of the implementation on these applications. The experiments compare versions of the algorithms instrumented using our library to standard un-instrumented versions.

Our approach to incremental computation works by first executing an *initial run* of the instrumented version on an initial input. This initial run builds all the structure needed for supporting updates (e.g., memo tables and dynamic dependences) and generates the output for the input. The experiments show that the performance of our instrumented code is often within a reasonably small factor of the standard un-instrumented algorithm. In the best case the instrumented code is 70% slower than the un-instrumented code. In the worst case it is 25 times slower. On average (over all our applications), the instrumented code is about 8 times slower than the un-instrumented code.

After the initial run, the input can be changed and *change propagation* is used to propagate the changes through the computation to update the output. This change-and-propagate process can be repeated as desired. To measure the effectiveness of change propagation, we study various forms of insertions and deletions into/from the input. The results demonstrate performance that greatly improves over re-running the un-instrumented algorithm from scratch. On average (over all our algorithms), change propagation performs better than running the un-instrumented code from scratch for inputs of size of greater than 11. For the basic routines on inputs of size 1,000,000, the time for change propagation is over 10,000 times faster than running the un-instrumented routine from scratch. For the computational geometry algorithms on inputs of size 100,000, the time for change propagation is between 100 and 3,000 (1,300 on average) times faster than running the un-instrumented standard algorithm from scratch. The asymptotic performance as input sizes grow matches the theoretical bounds proved elsewhere [2].

We note that the cost of garbage collection (GC) can be very high in our implementation—its allocation pattern is pessimal for a generational collector. We therefore give numbers both with and without GC costs.

2. Background

The problem of adjusting computations to changes has been studied extensively. Much of the previous work took place under the title of “incremental computation”. The most promising approaches have been centered on the idea of dependence graphs and memoization. This section reviews previous work on these techniques, points out their limitations, and describes how they can be combined. For a more comprehensive list of references to previous work, we refer the reader to Ramalingam and Rep’s bibliography [31].

2.1 Dependence Graphs

Dependence-graph techniques record the dependences between computation data. When the input is changed, a change-propagation algorithm is used to update the data affected by the change. Demers, Reps, and Teitelbaum [17] introduced static dependence graphs for the purpose of incremental computation. Static dependence graphs are not general purpose, because they do not permit the change-propagation algorithm to update the dependence structure of the computation.

Recent work generalized dependence-graph techniques by introducing *dynamic dependence graphs (or DDGs)* [4]. DDGs represent the dependences between data and the function that read them, and the dynamic containment relation between function calls. Given the DDG of a computation, any data can be changed, and the computation can be updated by performing change propagation. The change-propagation algorithm executes function calls that depend on the changed data, either directly or through changes made by other function calls. When executed, a function call can change some previously computed data, whose readers must then be executed. Further research on DDGs showed that they can support incremental updates as efficiently as special-purpose algorithms [6, 8]. DDGs can be applied to any functional (side-effect free) program. The performance of the technique, however, critically depends on the application.

To see a limitation of DDGs, consider the quicksort algorithm. Figure 1 shows the function-call tree of quicksort with the inputs $I = [11, 7, 4, 17, 5, 19, 9]$ (on the left) and $I' = [11, 7, 4, 17, 8, 5, 19, 9]$ (on the right). With DDGs, the change that transforms I to I' is performed by side-effecting memory to insert the new key 8. The change propagation algorithm re-executes all the function calls that directly or indirectly depend on the new

key. Figure 1 shows the function calls executed by change propagation (dotted sets). It has been shown that if an element is added to the end of the input list, the expected time for change propagation is $O(\log n)$ [4]. If the element is inserted in the middle of the list, however, change propagation will require $O(n)$ expected time, and if at the beginning it will require from-scratch execution [4]. DDGs give similar update times for many other applications. Even for simple list primitives such as `map`, `apply`, and `fold`, change propagation requires linear time on average (over all changes).

2.2 Memoization

The idea behind memoization [10, 26, 27] is to remember function calls and re-use them when possible. Pugh [29], and Pugh and Teitelbaum [30] were the first to apply memoization (also called function caching) to incremental computation. A key motivation behind their work was a lack of general-purpose technique for incremental computation (static-dependence-graph techniques that existed then applied only to certain computations [29]). As with DDGs, memoization can be applied to any functional (side-effect free) program, but the performance depends on the problem. Since Pugh and Teitelbaum’s work, other researchers investigated applications of memoization to incremental computation [1, 24, 25, 21, 5].

To see how memoization may aid in incremental computation, consider executing some program with some input I and then with a similar input I' . Since the inputs are similar, it may be that many of the function calls within the computations are applied with equivalent arguments. Thus, via memoization, many of function calls from the execution with I can be re-used during the execution with I' ; this can decrease execution time dramatically. Although this may sound intuitively true, it holds only for certain programs. In general, the effectiveness of memoization is limited.

To see a problem with memoization, consider again the quicksort algorithm shown in Figure 1. Even though the second input differs from the first by only one key, many function calls performed with the second input cannot be re-used from the first execution. In particular, the calls whose input contain the new key (8) cannot be re-used (Figure 1 highlight such calls). Similar problems arise in many other applications. Even with simple list primitives (e.g. `map`, `fold`, `apply`), incremental updates with memoization require linear time on average (over all changes).

2.3 Combining DDGs and Memoization

There is an interesting duality between memoization and DDGs: memoization identifies the parts of a computation that remain unaffected by a change and re-uses their results, while DDGs pinpoint the parts of the computation that are affected by the change and re-evaluates them. This duality can be seen concretely in Figure 1, where function calls not found by memoization and the function calls executed during change propagation intersect only at changes that are essential to update the output. Thus by combining these two techniques, it can be possible to execute only these essential nodes (marked in black in the figure).

Recent work showed that DDGs and memoization can be combined [2] to achieve this goal for quicksort and more generally for many other applications. The interface of a library that combines these techniques and some preliminary experimental results have also been described [3]. At a high-level, the idea behind the combination is to use memoization when running code during change-propagation. Since change propagation requires not only the results but also the DDG of a computation, memoization stores DDGs of function calls. Although this idea may seem natural, it is complex to support efficiently and correctly because of the interaction between DDGs and memoization. Section 3 describes efficient algorithms for combining DDGs and memoization.

3. Algorithms for Self-Adjusting Computation

This section describes efficient algorithms for combining dynamic dependence graphs and memoization. These algorithms form the basis for our implementation described in Section 4. We first present a high-level description of the combination and then give a precise specification of the algorithms.

3.1 The Main Ideas

We describe the language primitives for enabling the combination and give an overview of the algorithms for supporting these primitives and the critical invariants maintained by the algorithms.

The primitives. Our approach relies on extending an existing language with *modifiable references* or *modifiabls* for short. Modifiabls are memory locations that contain *changeable* values. Modifiabls are created by `mod` operations, and written by the `write` operations. The content of a modifiable can be accessed by the `read` operation. The `read` operation takes a modifiable and a function, called the *reader*, and applies the reader to the contents of the modifiable. Readers are only restricted to return no values. They are otherwise unrestricted: they can write to other modifiabls and they can contain other reads. This restriction enables tracking all data dependences on changeable values by tracking the `read` operations. Reader functions enable change propagation: when the contents of a modifiable m changes, the values that depend on that modifiable can be updated by re-executing the readers of m . To support memoization, we provide an operation called *lift* that takes a function f and two kinds of arguments, called *strict* and *non-strict*. When executed, a `lift` operation first performs a memo lookup by matching only the strict arguments. If the memo lookup succeeds, then the recovered computation is re-used by “adjusting” it to the non-strict arguments using change propagation. If the memo lookup fails, then the call is executed and the resulting computation is remembered.

The `mod`, `read`, `write`, `lift` operations can be organized in a type-safe interface that ensures the correctness of self-adjusting programs. We present a library that support such an interface elsewhere [3]. Since this paper focuses on the internals of the implementation, we do not address safety issues here.

DDGs, memo tables, and change propagation. As a self-adjusting program executes, its memo table, denoted by Σ , and its DDG, can be built by tracking the `mod`, `read`, `write`, and `lift` operations. A DDG consists of a set of modifiabls V , a set of reads R , a set of data dependences $D \subseteq V \times R$, and a containment hierarchy $C \subseteq R \times R$. The data dependences represent the dependences between modifiabls and reads: $(v \in V, r \in R) \in D$, if r reads v . The containment hierarchy C represents the control dependences: $(r_1, r_2) \in C$, if r_2 is contained within the dynamic scope of r_1 .

The `lift` operations populate the memo table Σ . Consider an execution of a `lift` operation with function f and the strict and non-strict arguments s and n , respectively. The operation first performs a memo look up, by checking if there is a call to f with the strict argument s in the *current computation context* or *context* for short. The context consists of a set of previously performed computations. If the lookup fails, then the function call is executed, and the computation is stored in the memo table Σ indexed by the function and the strict arguments. The stored computation includes the DDG of the computation and the memo-table entries created during the computation. If the lookup succeeds, the computation is retrieved and the non-strict arguments n are updated to their new values. Change propagation is then run on the computation. To facilitate change propagation, non-strict arguments themselves are treated as changeable values by placing them into modifiabls.

Given the DDG and the memo table for a self-adjusting program, any value stored in a modifiable can be changed by using the `write` operation, and the computation can be updated by running change propagation. The change-propagation algorithm maintains a queue of affected reads that initially contains the reads of the changed modifiables (a read is represented as a closure consisting of a reader and an environment). The algorithm repeatedly removes the earliest read from the queue, sets the context to the computation representing the previous execution of that read, and re-executes the read. By setting the context to contain computations from the previous execution of the read, the change-propagation algorithm makes these previously computed results available for re-use. When the change propagation algorithm terminates, the result and the DDG of the computation are identical to the result and the DDG obtained from a from-scratch execution.

Example. As an example of how this approach takes advantage of the duality of DDGs and memoization, consider the quicksort example shown in Figure 1. Inserting the new key **8** to the computation affects the first read containing the leftmost `sp` function drawn as a dark solid circle. The change-propagation algorithm therefore re-executes that function after placing all the calls contained in that function into the context for re-use. Since the function immediately calls `sp(5)`, and since that function is in the context, it is re-used. This process repeats until all affected read’s are re-executed (the re-executed reads and all new function calls are shown as dark circles). The amount of work performed is within a constant factor of the work that is necessary to update the computation.

Key invariants. A conflicting property of memoization and DDGs requires that certain invariants be maintained to ensure correctness: with DDGs, every function call in a computation is represented as a vertex in the DDG of that computation; with memoization, function calls and thus their DDGs can be re-used potentially multiple times. Since the change-propagation algorithm must return a well-formed DDG, and since a re-used DDG must be adjusted to changes both due to `write` operations and due to non-strict arguments, re-use of function calls must be restricted so that no function call is re-used more than once. Enforcing this invariant is complicated by the fact that when a function call v is re-used not only v but the descendants of v are re-used. In addition to this correctness invariant, effective change propagation requires that the execution order of all vertices be maintained. Original work on DDGs used constant-time order maintenance data structures for maintaining this order [4]. Since via memoization, function calls can be re-used in arbitrary order, their ordering becomes more difficult to maintain in constant time. Our approach therefore maintains the invariant that function calls are re-used in the same (relative) order as they were initially executed.

The approach ensures these invariants by carefully managing the content of the (current computation) context. During the initial run, the context is always kept empty. This ensures that no re-use takes place in the initial run, and therefore the resulting DDG is well-formed. During change propagation, before re-executing a function call v , the descendants of v are placed into the context—this makes them available for re-use. When a vertex v is re-used, the descendants of v and the vertices that come before v are removed from the context. Since the ancestors of a vertex are executed before that vertex, this ensures that the ancestors are also removed.

It is possible to relax these invariants for certain classes of computations. It is not known, however, if the invariants can be relaxed in general without increasing the asymptotic overhead of the algorithms for combining DDGs and memoization.

3.2 The Algorithms

This section describes the algorithms underlying our implementation. A key to the efficiency of the algorithms is the techniques for

representing memoized computations and enforcing the invariants the combination needs to maintain (Section 3.1) by using time intervals.

Intervals. Let $(U, <)$ be a totally ordered universe (set), where $<$ is an order relation. The universe U can be thought as a *virtual time line*, the elements of which are called the *time (stamps)*. In the rest of this section, we denote the time stamps by t (and variants) and, for the purposes of presentation, we do not mention the fact that time stamps are drawn from U . An interval δ is either empty or half-open. We say that the interval δ is half-open if the *stop time* $t_e(\delta)$ is included in δ while the *start time* $t_s(\delta)$ is not. The half open interval $(t_1, t_2]$, where $t_1, t_2 \in U$, is the set $\{t \in U \mid t_1 < t \leq t_2\}$. Throughout, we use lower case Greek letters μ, δ, π , and variants to denote intervals. As the names imply, the reader may find it helpful to think of U as the real line and the intervals as time intervals.

We say that a non-empty interval δ' is a *tail* of δ , denoted $\delta' \sqsubseteq \delta$, if $t_s(\delta) \leq t_s(\delta') < t_e(\delta') = t_e(\delta)$. If $\delta' \sqsubseteq \delta$ and $\delta' \neq \delta$, then δ' is a *proper tail* of δ , denoted $\delta' \subset \delta$. For a given set X of non-empty intervals, we say that the interval μ' is an X -*slice* of μ , if μ' is a proper tail of μ , and $\mu \setminus \mu'$ does not contain any start or stop time stamps of the intervals in X , i.e.,

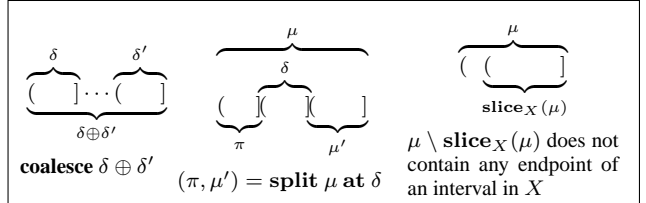
$$(\mu' \subset \mu) \wedge (\forall \delta \in X. (\delta \cap (\mu \setminus \mu') \neq \emptyset \Rightarrow (\mu \setminus \mu') \subset \delta)).$$

The algorithms require three operations on intervals, defined and illustrated as follows.

coalesce $\delta \oplus \delta'$, where δ and δ' are arbitrary intervals, is the interval $\{t \mid \exists t_1, t_2 \in \delta \cup \delta' \text{ s.t. } t_1 < t \leq t_2\}$.

split μ at δ , where $\mu \neq \emptyset$, $\delta \subset \mu$, and $\delta \not\sqsubseteq \mu$ yields a pair of intervals (π, μ') such that π, δ , and μ' are mutually disjoint and $\pi \cup \delta \cup \mu' = \mu$. Moreover, $\mu' \sqsubseteq \mu$ and $(\delta \cup \mu') \sqsubseteq \mu$. If $\delta = \emptyset$ then $\pi = \emptyset$ and $\mu' = \mu$.

slice $_X$ (μ) yields an X -slice of μ . This definition does not uniquely describe the result, but any X -slice will do.



DDGs and memo tables. Based on time intervals, we represent a DDG by the triplet (V, R, D) , where V is the set of modifiables, R is the set of reads, $D \subseteq V \times R$ is the set of data dependencies. We associate each read r with its interval, denoted $\Delta(r)$, with the *source* modifiable, denoted $\text{Src}(r)$, and the reader function $\text{Rd}(r)$. We define the Δ -operator on sets of reads point-wise as: $\Delta(R) = \{\Delta(r) \mid r \in R\}$. We say that a read r_1 is *contained* in another read r_2 if and only if the time interval of r_1 lies within the time interval of r_2 .

The memo table for the computation maps function calls to computations. A function call is represented as a function and the strict arguments of that function; a computation is represented as a triple consisting of the interval of the call, the modifiables for the non-strict variables, and the result of the call.

Execution and change propagation. Each expression (except for the “meta” operation `propagate`) executes in the context of an interval, called the *current interval*, denoted by μ (and its variants). The execution of an expression e in the context of μ , denoted $e \mu$, returns an updated current interval $\mu' \sqsubseteq \mu$, a fresh interval δ , and a result a , i.e.,

$$(\delta, \mu', a) \leftarrow e \mu, \text{ where } \mu' \sqsubseteq \mu \text{ and } \delta \cap \mu' = \emptyset \text{ and } \delta \cup \mu' \text{ is a } \Delta(R)\text{-slice of } \mu.$$

```

(* DDG is  $(V, R, D)$ 
 *  $Q$  is the priority queue
 *  $\Sigma$  is the memo table
 *)

mod ()  $\mu = V \leftarrow V \cup \{m\}$ , where  $m \notin \text{dom}(V)$ 
      return  $m$ 

read ( $m, f$ )  $\mu_0 =$ 
   $\mu_1 \leftarrow \text{slice}_R(\mu_0)$ 
   $(\delta_0, \mu_2, a) \leftarrow f(\text{Val}(m))\mu_1$ 
   $\mu_3 \leftarrow \text{slice}_R(\mu_2)$ 
   $(\delta, r) \leftarrow (\mu_1 \setminus \mu_3, [\text{Rd} \mapsto f, \text{Src} \mapsto m, \Delta \mapsto (\mu_1 \setminus \mu_3)])$ 
   $(R, D) \leftarrow (R \cup \{r\}, D \cup \{(m, r)\})$ 
  return  $(\delta, \mu_3, a)$ 

write ( $m, n$ )  $\mu =$ 
  if  $\text{Val}(m) \neq n$  then
     $(\text{update}(m, n) ; Q \leftarrow Q \cup \{r \mid (m, r) \in D\})$ 

propagate  $\delta =$ 
  while  $r \leftarrow \text{extractMin}_\delta(Q)$  do
     $(-, \mu, -) \leftarrow \text{Rd}(r)(\text{Val}(\text{Src}(r)))(\Delta(r))$ 
     $R \leftarrow \{r' \in R \mid \Delta(r') \not\subseteq \mu\}$ 
     $(D, Q) \leftarrow (D \upharpoonright_R, Q \upharpoonright_R)$ 

lift  $f(s, n)$   $\mu =$ 
  case  $\text{lookup}_{\Sigma, \mu}(f, s)$  of
    Found( $a, \delta, m$ )  $\Rightarrow (\pi, \mu') \leftarrow \text{split } \mu \text{ at } \delta$ 
       $R \leftarrow \{r \in R \mid \Delta(r) \cap \pi = \emptyset\}$ 
       $(D, Q) \leftarrow (D \upharpoonright_R, Q \upharpoonright_R)$ 
      write ( $m, n$ )
      propagate  $\delta$ 
      return  $(\delta, \mu', a)$ 
    | NotFound  $\Rightarrow m \leftarrow \text{mod}()$ 
      write ( $m, n$ )
       $(\delta, \mu', a) \leftarrow f(s, m)\mu$ 
       $\Sigma \leftarrow \Sigma[f(s, \cdot) \mapsto (a, \delta, m)]$ 
      return  $(\delta, \mu', a)$ 

```

Figure 2. An interval-passing semantics of the interface.

The expressions that correspond to `mod`, `read`, `write`, `lift` primitives treat intervals specially. All other expressions coalesce intervals while propagating the current interval forward. For example, e_1 and e_2 are sequentially executed as follows

$$\begin{aligned}
 (\delta_1 \oplus \delta_2, \mu'', a) &\leftarrow (e_1; e_2) \mu, \text{ where,} \\
 (\delta_1, \mu', -) &\leftarrow e_1 \mu \text{ and } (\delta_2, \mu'', a) = e_2 \mu'.
 \end{aligned}$$

Figure 2 shows the pseudo-code for the `mod`, `read`, `write`, `propagate`, and `lift` operations. The code globally maintains the $DDG = (V, R, D)$, and the memo table Σ , of the computation and a priority queue, Q , of affected reads. We note that, `mod` and `write` operations do not modify the current interval.

The `mod` operation extends the set of modifiables with a fresh modifiable m (that is not contained in the domain of V , i.e., $m \notin \text{dom}(V)$) and returns m .

The `read` operation starts by creating a time interval. The first `slice` ensures that each `read` has its own unique start time. The second `slice` guarantees that the interval $\Delta(r)$ is non-empty. Since

μ_3 is a tail of μ_1 , the `read`'s interval $\Delta(r) = \delta = \mu_1 \setminus \mu_3$ will be half-open.

The `write` operation checks if the value being written differs from the value stored in the modifiable. If so, the readers of the modifiable are inserted into the priority queue Q . Since the `write` operation does not involve reads, it does not perform any interval operations.

Change propagation (`propagate`) updates the given interval δ by repeatedly extracting the earliest affected `read` in δ from the priority queue and re-executing its reader. The re-execution takes place in the original interval of the read. This ensures that only the part of the virtual time line that belongs to the re-executed read is modified by the re-execution. When the re-execution is completed, the elements of R , D , and Q that do not belong to the new interval are expunged by restricting R , D , and Q to the new interval μ .¹ Change propagation for an interval δ stops when the queue contains no read operations that are contained within δ .

The `lift` operation takes a function, f , along with a strict argument s , and a non-strict argument n . A memo lookup seeks for a call to f that is contained within the interval μ , and whose strict argument is equal to s . When checking for the equality of the strict arguments, the lookup operation uses shallow equality where two locations are considered equal if they have the same address (or identity). If the lookup succeeds, then it returns a result a , an interval δ , and a modifiable m that contains the values of the non-strict argument. The algorithm extracts the interval δ from μ by using `split`, writes the new non-strict argument into m , and change-propagates into δ . This adjusts the re-used computation to all changes. The algorithm then expunges the elements that belong to the interval π and returns. This ensures that all elements of the DDG that do not belong to the current computation are removed. If the memo lookup is not successful, then the algorithm creates a new modifiable m and writes the non-strict argument into m , and applies f to s and m . Finally the algorithm remembers a , m , and δ in the memo table Σ .

Restricting the memo look ups to the current interval, and sliding the current interval past the interval of the re-used computation, δ , (by splitting the current interval at δ) ensures that each function call is re-used at most once. As mentioned in Section 3.1, this is critical for correctness. Sliding the current interval past the re-used interval also ensures that the ordering between the intervals of the function calls respect their existing ordering. This enables implementing the time line using constant-time data structures (Section 4).

4. Implementation

We implemented the approach described in Section 3 as a library for the Standard ML language. This section gives an overview of the main data structures, discusses the space integrity property (Section 4.2), and describes some optimizations (Section 4.3). The full code for the library can be found at

<http://ttic.uchicago.edu/~umut/sting/>

4.1 Data Structures

Intervals. The implementation globally maintains a (*virtual*) *time line* that consists of a totally ordered set of *time stamps*. An interval is represented as a pair of time stamps, e.g., the interval $(t_1, t_2]$ is represented with the the pair (t_1, t_2) . Three operations are used to maintain the time line: the `insert` operation inserts a new time stamp immediately after a given time stamp in the time line; the `delete` operation removes a given time stamp from the time line;

¹ The notation $X \upharpoonright_R$ denotes the restriction of X to elements that do not mention reads outside R .

and the `compare` operation compares the ordering of time stamps in the time line. In addition to the time line, the implementation maintains two time stamps, called the *current time* and the *finger*. Together these define the *current interval*.

The `insert`, `delete`, and `compare` operations suffice to support all operations on intervals (Section 3.2). The `slice` operation is implemented by inserting a new time stamp t after the current time and setting the current time to t . The `split` operation is implemented by deleting all time stamps between the current time and the desired interval. Since the implementation of the `slice` operation advances the current time, an explicit use of `coalesce` operations does not arise.

Since our algorithms operate on the time line extensively, maintaining it efficiently is critical for performance. We therefore implemented the (amortized) constant-time order maintenance data structure of Dietz and Sleator [18] (our implementation is based on Bender *et al.*'s description [11]).

Dynamic dependence graphs. The implementation globally maintains the current dynamic dependence graph (DDG). Each read consists of a closure and a pair of time stamps representing its interval. Each modifiable reference is implemented as a reference to a tuple consisting of a value and a read list. A *read list* is maintained as a doubly linked list of reads.

Lift functions and memo tables. The implementation treats lift functions somewhat differently from the algorithmic description (Section 3).

The first difference concerns the maintenance of memo tables. Instead of maintaining one memo table for the whole computation, the implementation provides a primitive, called `mkLift`, for creating lift functions that have their own memo tables. The `mkLift` primitive allocates a memo table and specializes a generic `lift` function for the allocated memo table; the specialized function uses its own memo table for all memo operations. This design decision obviates the need for comparing functions.

The second difference is in the association of time intervals with memo entries. In the algorithmic description (Section 3), a memo entry can have an empty interval. In particular, if the memoized computation does not perform any reads, then the interval for that computation will be empty. This makes it impossible to determine to if a memo entry is live, *i.e.*, belongs to the current computation. Therefore, it is possible for memo tables to accumulate over time. In order to ensure strict bounds on the space consumption, the implementation associates a non-empty interval with each memo entry.² To achieve this, when a memo entry is first created, the implementation checks if the interval for that entry is empty. If not, then the implementation proceeds as usual. If the interval is empty, then the implementation creates two consecutive time stamp, t_1 and t_2 , following the current time, and assigns the interval $(t_1, t_2]$ to the memo entry. This makes it possible to delete memo entries that are not live by checking whether t_1 (or t_2) is live—Section 4.2 describes how this can be used to ensure a space integrity property.

Memo tables are implemented as hash tables with chaining [23]. A hash table is an array of buckets, each of which points to a list of *entries*. Each entry represents a function call and consists of the result, the time interval (two time stamps), and the non-strict arguments of the call. Memo lookups are performed by hashing the strict arguments of the call. A lookup succeeds if there is a memo entry within the current interval that has the same strict arguments.

² An alternative is to maintain memo tables as caches with a particular cache replacement policy. Unfortunately, no cache-replacement policy can be optimal for all computations—for any given policy there will be applications for which the choice is suboptimal. Cache replacement policies can also make it difficult to prove asymptotic time bounds and cause unpredictable performance in practice.

Insertion and deletions operations are supported as usual. The implementation ensures that the *load factor* of the hash tables do not exceed two by doubling the tables as necessary.

For fast equality checks and hashing, the implementation relies on boxing (*a.k.a.*, tagging). Every strict argument to a lift function is required to be tagged with a unique identity (an integer) (since ML is type safe, values of different types can have the same identity). These tags are used both for computing the hashes, and for resolving collisions.

4.2 Space Integrity

The implementation ensures that the total space usage never exceeds the space that would be used by a from-scratch execution of the (self-adjusting) program with the current input. We refer to this property as *space integrity*. The property implies that the space usage is independent of the past operations (*i.e.*, history). As an example, consider running a program P with some input and then performing a sequence of change-and-propagate steps, where each step makes some change and runs change propagation. At the end of these operations, the total space usage is guaranteed to be the same as the space used by a from-scratch execution of P with the final input.

The implementation enforces space integrity by eagerly releasing all references to trace elements, (modifiables, reads, memo entries, time stamps) that do not belong to the current computation. This makes the trace elements available for garbage collection as soon as possible. Since modifiable references are only referenced by reads and memo entries, and since reads, memo entries, time stamps are not referenced by any other library data structures, releasing the reads, memo entries, and time stamps suffices to ensure space integrity.

A key property of the implementation is that each *live* read and memo entry (which belong to the current computation) have an associated time stamp. The implementation relies on this property to ensure space integrity. The idea is to maintain conceptual *back pointers* from each time stamps to the reads and memo entries with which the time stamp is associated. Note that each memo entry and each read are associated with two time stamps (a start and an end time stamp). Since each read and memo entry should be released once, it suffices to maintain back pointers only at one time stamp. In the implementation, we choose to keep the back pointers for reads at their start time stamps, and the back pointers for memo entries at their end time stamps. As an example, consider a time stamp t being deleted (by the `split` operation). First, t is removed from the order maintenance data structure. Second, the back pointers of t are used to find the read or the memo entries associated with t . Third, the read is removed from the read list that contains it, or all associated memo entries are removed from the memo tables that they are contained in. Due to some subtle constraints of the type system of the Standard ML language, we implement the back pointers by associating a closure, called the *release function*, with each time stamps. When a time stamp is deleted, its release closure is executed. Since multiple memo entries can have the same end time stamp, a time stamp can have multiple release functions—these are represented by composing them into one function.

We have found that the space-integrity property is critical for the effectiveness of the library. Our earlier implementations suffer from space explosion, because they lazily release reads and memo entries (by flagging deleted reads and memo entries and releasing flagged objects when next encountered).

To verify that our implementation ensures the space-integrity property, we implemented a space-profiling version of our library. Using the library, we experimentally verified that the numbers of all live library-allocated objects (modifiables, reads, time stamps, memo entries, and memo tables) after various sequences of change-

and-propagate steps are equal to the numbers obtained by from-scratch execution.

4.3 Optimizations

We describe an optimization that we employed in implementing the library and two specialized primitives for reading modifiables and creating lift functions that take advantage of specific usage patterns. To use the optimized primitives, the programmer replaces each call to an operation with a call to the optimized version of that operation. This may require making some small changes to the code.

Single reads. In many applications, most modifiables are read only once. For example, for all the applications considered in this paper, the average number of reads per modifiable is less than 1.5. To take advantage of this property we implemented a version of the read lists data structure described in Section 4.1 that is specialized to contain no extra pointers when the list consists of only one read. When the list contains multiple reads, the data structure separates one of the reads as special, and places all other reads in a doubly linked list. The first element of the doubly linked list points to the special list and the second element. This data structure can be thought as a doubly linked list that has two base cases, an empty and a single-read case.

This optimization makes the operations on reads lists slightly more complicated than a standard doubly-linked lists and also complicates the eager deletion of reads—when a read becomes the only read, then its release closure may need to be updated to account for this change. Since a large percentage of the modifiables are read only once, however, we found that this optimization can improve the running times and reduce the size of the memory print. We also experimented with read lists optimized for both single and double reads but observed little or no improvement over the single-read optimization.

Immediate reads. This optimization concerns the elimination of *immediate reads* whose time intervals are empty. Sometimes, a modifiable reference is written and is then immediately read in order to copy its value to another modifiable. A modifiable reference sometimes performs a non-self-adjusting computation that creates no time stamps (other than the times stamps that would be created for the read itself). For example, a function f may call some other function g , which writes its result to a modifiable reference. Function f can then read the result computed immediately after its written, place the value read in a tuple, and write it to another modifiable. Since such reads take place immediately after the write and since they contain no other time-stamp creating computations, they need not be associated with a proper time interval—they can be piggybacked to the preceding write. When a modifiable is written, the piggybacked read is executed immediately. Beyond replacing the library-provided `read` function with the immediate read function, this optimization requires no further changes to existing code. We used this optimization in the implementation of `lift` operations. For our benchmarks, we have observed that this optimization can reduce the running time by up to 10% for both from-scratch runs and change propagation.

Inlining lift operations. This optimization eliminates modifiables and reads that are created by the lift operations due to non-strict arguments when the non-strict arguments are only passed to a tail call. The idea is to store the value of the non-strict arguments directly in the memo table. When a memo match takes place, the recursive tail call is performed explicitly. This optimization can be viewed as an inlining of the lift operation; it saves a modifiable, a read, and two time stamps.

As an example, consider the call $f(s, n)$ to a lift function f with the strict argument s and a non-strict argument n . When called,

the function first performs a memo look up using s . If there is a memo match, then the match yields a result r and a modifiable m that stores the value of the non-strict argument for the re-used call. Before returning the result, the function writes n into m and performs change propagation. This change propagation adjusts the computation according to the value the non-strict argument n . If n was just passed as argument to a tail call, then change propagation simply executes that call. With the optimization, the tail call is executed explicitly instead of relying on change-propagation.

To support this optimization, the library provides a version of the `mkLift` primitive that takes the function to be tail called as a separate argument. To use the optimization, the programmer specifies the tail call explicitly. The opportunity for this optimization arises often. In our benchmarks, we have observed that the optimization can reduce the running time for both change propagation and from-scratch execution up to 40% depending on the application.

4.4 Garbage Collection

Self-adjusting computation contradicts some key assumptions that generational garbage collectors rely on.

1. Due to the side-effecting nature of `write` operations, it is common for old data to point to new data, and
2. A significant portion of the total allocated data, i.e., trace data (modifiables, reads, memo table entries, and time stamps), have long lifespans.

The generational garbage collection systems that we experimented with (MLton, SML/NJ) do relatively poorly when the total amount of live trace data becomes large compared to the available memory (Section 6). We discuss some possible directions for better garbage collection support in Section 7.

4.5 Asymptotic Performance

We show elsewhere [2] that the overhead of the implementation described here is (expected) constant (the expectation is taken over internal randomization used for hash tables). This implies that a from-scratch execution of a self-adjusting program is by a constant factor slower than the from-scratch execution of the corresponding non-self-adjusting program.

In addition to the overhead, asymptotic complexity of change propagation (under a given change) can also be analyzed using algorithmic techniques. Acar's thesis develops an analysis technique called *trace stability* for this purpose and gives tight bounds on some of the applications considered here [2]. At a high level trace stability measures the edit-distance between the traces of the program with two inputs. For a particular class of computations called *monotone*, it is shown that the time for change propagation is bounded by the edit distance. Informally, we say that a program is *stable* if the edit-distance between its traces is small under small changes.

5. Applications

To evaluate the effectiveness of the approach, we implemented standard and self-adjusting versions of the following applications.

- **filter:** Takes a list and a boolean function and returns the list of values from the list that satisfy the function.
- **fold:** Takes a list and an associative binary operator and applies the operator to the elements of the list to produce a single value.
- **map:** Takes a list and a function and constructs another list by applying the function to each element.
- **reverse:** Reverses a list.

- **split**: Takes a list and a boolean function and returns two lists consisting of the elements for which the function returns `true` and `false` respectively.
- **quick-sort**: The quick-sort algorithm for list sorting.
- **merge-sort**: The randomized merge-sort for list sorting.
- **graham-scan**: The Graham-Scan convex-hull algorithm [20].
- **quick-hull**: The quick-hull algorithm for convex hulls [9].
- **merge-hull**: The merge-hull algorithm for convex hulls.
- **ultimate**: A randomized version of Chan’s convex-hull algorithm [14].
- **diameter**: Shamos’s algorithm for finding the diameter of a set of points [33].

These applications are chosen to span a number of computing paradigms including simple iteration (`filter`, `map`, `split`), accumulator passing (`reverse`, `quick-sort`), incremental result construction (`graham-scan`), random sampling (`fold`), and divide-and-conquer (`merge-sort`, `quick-sort`, `merge-hull`, `ultimate`). The `graham-scan` algorithm combines a convex-hull algorithm with a linear scan. Similarly, the `diameter` algorithm combines a convex-hull algorithm with a linear scan. For some of the problems considered here, we show elsewhere [2] asymptotic bounds that closely match the best bounds achieved by special-purpose algorithms developed in the algorithms community.

List primitives. The list primitives consists of `filter`, `fold`, `map`, `reverse`, `split`. Except for `fold`, all of these applications perform a traversal of the list as they construct the output iteratively. To implement `fold`, we use a random-sampling technique instead of the usual iterative approach. The self-adjusting version of the iterative algorithm does not yield an efficient self-adjusting algorithm, because it is not stable—a small change to the input can cause a nearly complete re-computation in the worst case. The random sampling approach ensures stability by braking symmetry of the input randomly. The algorithm computes the result by randomly sampling the list and applying the operation to this sample. The algorithm repeats this sampling step until the list reduces to a single element. For all these algorithms, it is reasonably straightforward to show constant stability and hence, constant-time update bounds for small changes (e.g, an insertion or deletion).

Sorting algorithms. The sorting algorithms `quick-sort` and the randomized `merge-sort` algorithms are standard. The `quick-sort` algorithm divides its input into two sublists based on the first key in input (selected as the pivot) and recursively sorts each half. The algorithm avoid concatenating the results by passing the sorted half in an accumulator. The randomized `merge-sort` algorithm divides its input into two sublists by randomly selecting for each element in the input a destination sublist. The sublists are then sorted recursively and merged as usual. The deterministic version of the `merge-sort` algorithm can also be made self-adjusting but it is not as stable as the randomized algorithm, which is within an expected constant factor of the optimal for an insertion or deletion [2]. Both `quick-sort` and `merge-sort` use the `split` primitive to partition their input into two sublists.

Computational-geometry algorithms. The computational-geometry algorithms consists of a number of convex-hull algorithms (`graham-scan`, `qhull`, `merge-hull`, `ultimate`) and an algorithm for computing the diameter of a point set (`diameter`). The convex hull of a set of points is the smallest polygon enclosing the points. The *static* convex hull problem requires computing the convex hull of a static (unchanging) set of points. The *dynamic* convex hull problem requires the maintenance of the convex hull of a set of points, as the point set changes due to insertions and deletions. Both the static [20, 22, 14, 9, 34] and the dynamic [28, 32, 15, 12]

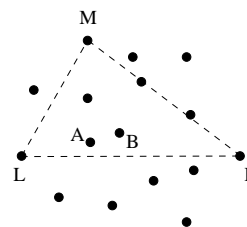


Figure 3. Chan’s algorithm for convex hulls.

convex hulls have been studied extensively for over two decades. To appreciate the complexity of the dynamic convex-hull problem, it should suffice to look over any of the aforementioned papers.

Of the convex hull algorithms that we consider, Chan’s algorithm is the most sophisticated. The algorithm is optimal in the size of the output (not just the input). This divide-and-conquer algorithm performs a special elimination step before the recursive calls. The elimination step removes a constant fraction of points from the inputs to recursive calls—it is the key to the optimality of the algorithm. Figure 3 illustrates how a randomized version of the algorithm works. For symmetry reasons, we describe only how the upper half of the hull is constructed. Given the leftmost (L) and the rightmost (R) points, the algorithm picks a random pair of points (A, B), and finds the farthest point (M) from the line (A,B). The algorithm then pairs the points randomly and eliminates an expected constant fraction of the points by applying an elimination test to each point. The algorithm then computes the left and right halves of the problem defined by the extreme points (L,M) and (M,R) respectively.

The diameter of a set of points is the distance between the pair of points that are farthest away from each other. By definition, such a pair is on the convex hull of the point set. For computing the diameter, we use Shamos’s algorithm [33]. This algorithm first computes the convex hull the points and then performs an iterative traversal of the convex hull while computing the maximum distance between anti-podal points. In our implementation of the `diameter`, we use the `quick-hull` algorithm to compute the convex hull.

All the computational geometry algorithm rely on the lists primitives to perform various functions. The `ultimate` and `quick-hull` applications use `fold` to find the point furthest away from a given line. The `merge-hull` application relies on the `split` primitive to divide its input into two sublists randomly, recursively computes their convex hulls and the merges the two hulls. The `graham-scan` application relies on a sorting step to sort the input points with respect to their x -coordinates (we use the `merge-sort`).

6. Experiments

This section describes an experimental evaluation of the implementation (Section 4) with benchmarks derived from the our applications (Section 5). We report detailed results of one of the computational geometry algorithms—the `ultimate` convex-hull algorithm—and summarize results for the rest of the benchmarks.

6.1 The Benchmarks

For the experiments, we implemented a *static* (non-self-adjusting) version, and a *self-adjusting version* of each application described in Section 5. To give a sense of how the two versions differ, Table 1 shows the number of lines and the number of tokens for the static and the self-adjusting versions of our sorting and convex-hull benchmarks (as counted by `wc` utility).³ The self-adjusting

³Since the list primitives rely on library functions, they are not included. A “token” is a string of characters delimited by white spaces.

Application	Static # Lines	Self-Adj. # Lines	Static # Tokens	Self-Adj. # Tokens
merge-sort	94	109	262	335
quick-sort	47	62	152	215
graham-scan	125	132	383	427
merge-hull	203	212	508	676
quick-hull	117	126	405	425
ultimate	207	208	630	697
diameter	177	199	558	660

Table 1. Number of lines and tokens for some applications.

versions contain about 10% more lines and 20% more tokens their static versions (on average). Much of this effectiveness is due to the ability to compose (or combine) functions as with conventional programming (as described in Section 5, the applications rely on other applications, especially list primitives, to compute various intermediate results).

Except for the `fold` application, the underlying algorithms for both versions are identical—the ordinary version of `fold` uses the `foldr` primitive provided by the SML Basis Library. All our benchmarks are derived by specializing the applications for particular inputs. The `filter` benchmark filters out the even elements in a list. The `map` benchmark maps an integer list to another integer list by adding a fixed value to each element in the list. The `minimum` benchmark computes the minimum of a list of integers. The `sum` benchmark sums the floating-point numbers in a list. Both `sum` and `minimum` are applications of the `fold` primitive. The `reverse` benchmark reverses a list of integers. The `split` benchmark divides inputs input into two lists based on a random hash function that maps each element to true or false. The sorting benchmarks, `quick-sort` and `merge-sort` sort strings. All convex-hull and the `diameter` benchmarks compute the convex hull and diameter of a set of points in two dimensions respectively.

6.2 Input Generation

We generate the input data for our experiments randomly. To generate a list of n integers, we choose a random permutation of the integers from 1 to n . To generate a floating-point or a string list, we first generate an integer list and then map the integers to their floating point or string representation (in decimal). For the computational-geometry algorithms, we generate an input of n points by picking points uniformly randomly from within a square of size $10n \times 10n$. The dynamic changes that we consider are described in the next section.

6.3 Measurements

We ran our experiments on a 2.7GHz Power Mac G5 with 4GB of memory. We compile our benchmarks with the MLton com-

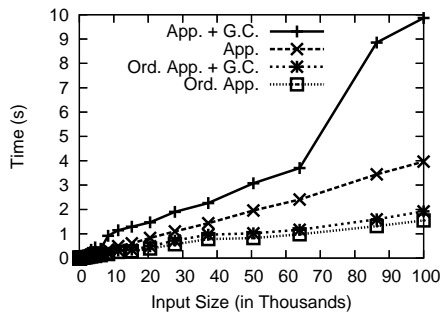


Figure 4. Time for initial-run (seconds) versus input size for `ultimate`.

piler using “`-runtime ram-slop 1 gc-summary`” options. The “`ram-slop 1`” option directs the run-time system to use all the available memory on the system—MLton, however, can allocate a maximum of about two Gigabytes. The “`gc-summary`” option directs the run-time system to collect summary information about garbage collection. In particular, the system reports the percentage of the total time spent for garbage collection.

In our measurements we differentiate between application time and total time. The *total time* is the wall-clock time, the *application time* is the total time minus the time spent for garbage collection. We measure the following quantities.

- **Time for from-scratch execution:** The time for executing the ordinary or the self-adjusting versions from-scratch.
- **Average time for an insertion/deletion:** This is measured by applying a delete-propagate-insert-propagate step to each element. Each such step deletes an element, runs change propagation, inserts the element back, and runs change propagation. The average is taken over all propagations.
- **Crossover point:** This is the input size at which the average time for change propagation becomes smaller than the time for the from-scratch execution of the ordinary program.
- **Overhead:** This is the ratio of the time for the from-scratch execution of the self-adjusting version to the time for the from-scratch execution of the ordinary version with the same input.
- **Speedup:** This is the ratio of the time for the from-scratch run of the ordinary version to the average time for insertion/deletion.

6.4 The Results

Figures 4, 5, 6 and 7 show various measurements with Chan’s algorithm (`ultimate`). Figure 4 compares the from-scratch runs of the ordinary and self-adjusting versions. This figure shows that the overhead of the self-adjusting version over the ordinary version is about a factor of five including GC time; without GC, the overhead is less than a factor of three. Figure 5 shows the average time for change propagation after an insertion/deletion. When GC time is excluded, the time grows very slowly with input size. When including the GC time, the time grows faster (especially towards the larger end of the horizontal axis) as the ratio of the size of the trace data to the size of the available space increases. Figure 6 shows the average speedup (the time for recomputing from scratch divided by average time for change propagation). Without the GC time, the speedup increases nearly linearly with the input size. When including GC time, the speedup grows more slowly. Still, when the input size is 100,000, change propagation is more than a factor of 600 faster than recomputing from scratch. Figure 7 shows the number of modifiables (modrefs), reads, and number

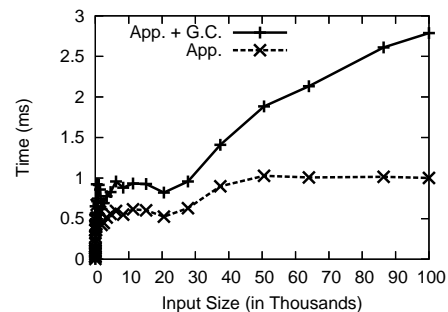


Figure 5. Average time for insertion/deletion (milliseconds) versus input size for `ultimate`.

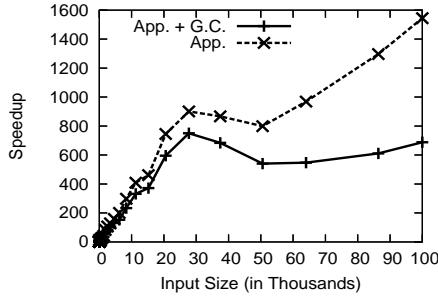


Figure 6. Speedup versus input size for ultimate.

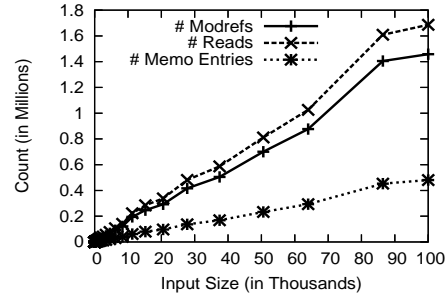


Figure 7. Space detail versus input size for ultimate.

Application	n	Ord./Run		Self/Run		Self/Propagate		Overhead		Crossover		Speedup	
filter	10^6	1.9	2.5	3.2	5.1	5.3×10^{-6}	1.4×10^{-5}	1.7	2.1	1	1	$> 4 \times 10^5$	$> 2 \times 10^5$
sum	10^6	1.0	1.2	3.7	6.4	8.3×10^{-5}	1.1×10^{-4}	3.8	5.5	1	1	$> 10^4$	$> 10^4$
map	10^6	1.8	2.5	3.9	12.6	4.8×10^{-6}	1.2×10^{-5}	2.1	5.1	1	1	$> 4 \times 10^5$	$> 2 \times 10^5$
minimum	10^6	1.8	2.4	3.7	6.4	1.6×10^{-5}	3.4×10^{-5}	2.1	2.6	1	1	$> 10^5$	$> 7 \times 10^4$
reverse	10^6	1.8	2.4	4.7	13.8	5.0×10^{-6}	1.6×10^{-5}	2.6	5.8	1	1	$> 4 \times 10^5$	$> 10^5$
split	10^6	2.2	2.9	2.9	4.9	6.2×10^{-6}	1.4×10^{-5}	1.3	1.7	1	1	$> 4 \times 10^5$	$> 2 \times 10^5$
quick-sort	10^5	0.4	0.5	3.1	11.2	3.8×10^{-4}	1.1×10^{-3}	8.5	23.2	1	1	> 955.4	> 431.3
merge-sort	10^5	1.2	1.4	6.4	13.8	3.8×10^{-4}	1.3×10^{-3}	5.2	9.7	1	1	$> 3 \times 10^3$	$> 10^3$
graham-scan	10^5	1.6	1.8	9.8	43.3	8.0×10^{-4}	3.2×10^{-3}	6.2	24.1	29	52	$> 2 \times 10^3$	> 557.3
merge-hull	10^5	2.1	2.4	7.7	44.9	5.9×10^{-3}	1.9×10^{-2}	3.7	18.9	39	52	> 352.7	> 124.0
quick-hull	10^5	1.0	1.3	3.2	8.8	2.1×10^{-4}	4.0×10^{-4}	3.2	6.8	17	17	$> 5 \times 10^3$	$> 3 \times 10^3$
ultimate	10^5	1.5	1.9	4.0	9.9	1.0×10^{-3}	2.8×10^{-3}	2.6	5.1	1	1	$> 2 \times 10^3$	> 688.0
diameter	10^5	1.6	1.8	3.1	7.8	2.5×10^{-4}	4.4×10^{-4}	2.0	4.2	10	13	$> 5 \times 10^3$	$> 4 \times 10^3$

Table 2. Measurements with our benchmarks (all times are in seconds).

of memo entries allocated for various input sizes. The number of modifiables and reads closely follow the execution time. Note also that the number of reads is only slightly larger than the number of modifiables. For our applications, we observed that modifiables are read slightly more than once on average.

Table 2 shows the measurements with all our benchmarks at fixed input sizes (the column titled “n” specifies the input size). The columns are divided into two sub-columns (except for the “n” column) that report the measured quantity excluding and including the time for garbage collection respectively. The “Ord. Run” column shows the time for a from-scratch execution of the ordinary version, the “Self-Adj. Run” column shows the time for a from-scratch execution of the self-adjusting version, the “Change Propagate” column shows the average time for change propagation after an insertion/deletion. The “Overhead”, “Crossover”, and the “Speedup” columns show the corresponding quantities, as defined in Section 6.3. For example, with the `filter` benchmark the ordinary version takes 1.9 and 2.5 seconds without and with the GC time respectively; change propagation takes 5.3×10^{-6} seconds and 1.4×10^{-5} seconds without and with the GC time; the overhead is a factor of 1.7 when the time for GC is excluded and 2.1 when the time for GC is included; and the speedup is more than factor of 4×10^5 and 2×10^5 without and with GC time.

The table shows that, when the GC time is excluded, overheads are well within a factor of ten for all applications (the average is 3.5). When the time for garbage collection is included, over-

heads are higher (the average is 8.8). What determines the overhead is the ratio of the total cost of the library operations (operations on modifiable reference and memoization) to the amount of “real” work performed by the ordinary algorithm. The overhead for the list primitives is relatively small, because these operations perform relatively few library operations. For the sorting applications, overheads are higher, because comparisons are cheap. Between two sorting algorithms, the overhead for mergesort is lower because mergesort is more memory intensive (due to the separate split and merge steps). For our computational geometry applications, the overheads are smaller, because geometric tests are relatively expensive. One exception is the Graham Scan algorithm, where the dominating cost is the sorting step.

Although the overhead is an important quantity, it should not be overemphasized, because it pertains to a from-scratch execution. With a self-adjusting application, from-scratch executions are rare, because changes to the input can be handled by change propagation. To measure the effectiveness of change propagation compared to from-scratch execution, we measured the crossover point (where change propagation becomes cheaper than recomputing), and the speedup. As the table shows, the crossover point under a single insertion/deletion is small for all our applications, and change propagation leads to orders of magnitude speed up over recomputing from scratch.

The reason for high speedups obtained by change-propagation is that there is often a near-linear time asymptotic gap between re-

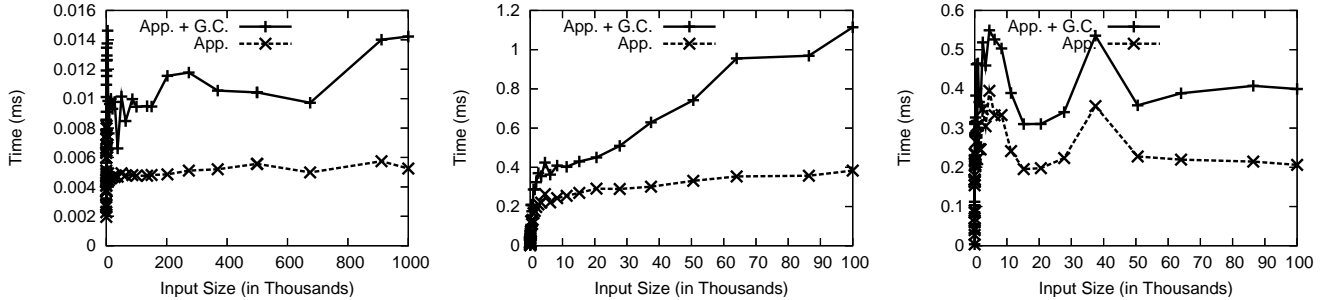


Figure 8. Average time (milliseconds) for insert/delete with filter, quick-sort, and quick-hull (from left to right) versus input size.

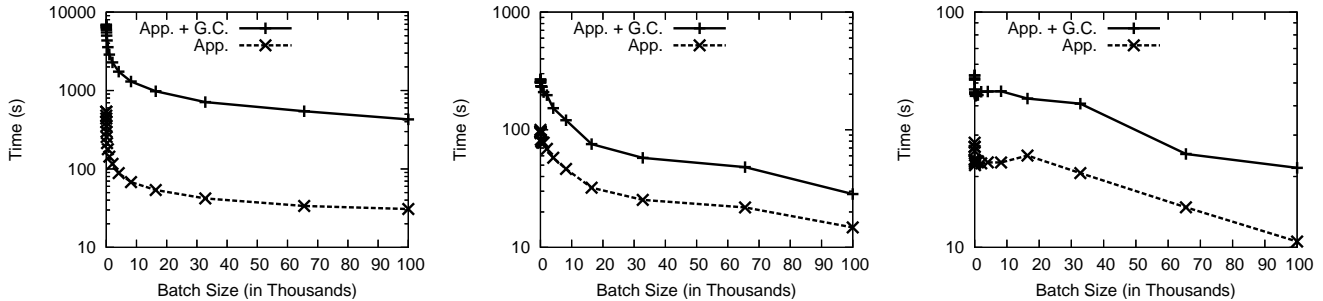


Figure 9. Time (seconds) for batch changes with merge-hull, ultimate, and diameter versus batch size.

computing from scratch and change propagation. The first author’s thesis proves this claim for a number of applications [2]. Our experiments validate this claim. Figure 8 shows the average time for change propagation after an insertion/deletion for a sample of our benchmarks. As can be seen, the application time grows slowly as the input size increases (especially when not including the GC time). The timings for other applications are similar.

6.5 Batch Changes

An important property of the change-propagation algorithm is its generality: it can adjust a computation to any change. For example, changes can be performed simultaneously in batches. We present some experimental results that show that batching changes can be significantly faster than performing them one by one.

To determine the effectiveness of batching, we perform the following experiment: We move all points in the input to a new location (by multiplying the coordinates of each point with a constant less than one) in varying sizes of batches. Each batch moves a number of points and performs a change propagation. We measure the average time for moving one point by dividing the total time it takes to move all points divided by the number of points.

Figure 9 shows the timings with the merge-hull, ultimate, and the diameter applications. The y axis is the running time in logarithmic scale, the x axis is the batch size. A measurement (x, y) means that the time for replacing each point (p) in the input (with $0.8 \cdot p$) while performing change propagation after every x replacements is y . For all applications, the input size is fixed at 100K. The figure shows that the running time for merge-hull and ultimate drops sharply with batch size (note the logarithmic scale). With diameter, the gains are less dramatic but still significant. The performance benefits of performing changes in batches is determined by the amount of interaction between the changes—the more they interact, the more the performance gains. When the batch size is equal to the input size, each point is replaced and change propaga-

tion is run only once. At this point, the total time is about a factor of three more than time for a from-scratch execution.

7. Discussions

Overhead and speedup. We believe that the overhead of our approach and the speedups can be further improved by direct compiler support for self-adjusting-computation primitives. With compiler support, we can give low-level implementations for our key data structures such as modifiables, reads, read lists, time stamps. These implementations would be significantly faster and consume less space by avoiding extra indirections that arise in the Standard ML language.

Garbage collection. Since trace elements (modifiables, reads, memo tables etc.) are known to have a long life span, and since the change-propagation algorithm explicitly frees the parts of the trace that become garbage (this is critical for the correctness of the algorithm), we expect that it will be possible to design a garbage collection algorithm that can deal with traces efficiently.

Correctness. The algorithms for combining memoization and dynamic dependence graphs are complex. It is quite difficult to argue about their correctness even informally. The combination, however, can be formalized by giving an operational semantics [2]. We expect that it will be possible (but difficult) to prove the correctness of the algorithms based on this operational semantics. Such a proof would also elucidate various invariants maintained by the operational semantics formally and can give insight about whether they can be relaxed.

Other applications. This paper evaluates performance of self-adjusting programs under so called *discrete changes* where input elements are deleted and inserted. Self-adjusting programs can also adjust to so called *continuous changes* (e.g., due to motion in geometric algorithms). In fact, the computational-geometry applications described here can automatically maintain their output under

motion when linked with a *kinetic library*. A description of such a kinetic library can be found elsewhere [7, 2]. The code for the kinetic library and the kinetic versions of our applications is available at <http://ttic.uchicago.edu/~umut/sting/>

Alternative combinations. This paper focuses on a particular combination of dynamic dependence graphs and memoization that ensure constant-time overheads. To achieve this performance guarantee, this combination puts certain restrictions on computation re-use. For a class of computations, called *monotone*, the combination guarantees maximal result re-use [2]. For non-monotone computations, we expect that it is possible to provide for more flexible result re-use by giving other combinations of dynamic dependence graphs and memoization—especially if one is content with logarithmic (instead of constant) overheads.

8. Conclusion

Our experiments demonstrate that the combination of DDGs and memoization is effective in practice for a variety of algorithms. This required some care in the implementation; the paper described the key techniques that were necessary. We believe this is the first experimental evidence that incremental computation of any type is effective in practice for a reasonably broad set of algorithms and computational paradigms (divide-and-conquer, random sampling, accumulator passing, and incremental construction).

Acknowledgments

We thank Jorge Vites for his help with implementing the computational geometry algorithms and Matthew Hammer for his feedback.

References

- [1] M. Abadi, B. W. Lampson, and J.-J. Levy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.
- [2] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.
- [3] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. A library for self-adjusting computation. In *ACM SIGPLAN Workshop on ML*, 2005.
- [4] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.
- [5] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.
- [6] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vites, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.
- [7] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and J. Vites. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.
- [8] U. A. Acar, G. E. Blelloch, and J. L. Vites. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*, 2005.
- [9] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.
- [10] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- [11] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Lecture Notes in Computer Science*, pages 152–164, 2002.
- [12] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 617–626, 2002.
- [13] M. Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 26–35. ACM Press, 2002.
- [14] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996.
- [15] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48(1):1–12, 2001.
- [16] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.
- [17] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.
- [18] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 365–372, 1987.
- [19] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.
- [20] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.
- [21] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on PLDI*, pages 311–320, May 2000.
- [22] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm. *SIAM Journal on Computing*, 15(2):287–299, 1986.
- [23] D. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison Wesley, 1998.
- [24] Y. A. Liu, S. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 157–170, 1996.
- [25] Y. A. Liu, S. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.
- [26] J. McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.
- [27] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.
- [28] M. H. Overmans and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.
- [29] W. Pugh. *Incremental computation via function caching*. PhD thesis, Department of Computer Science, Cornell University, August 1988.
- [30] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.
- [31] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 502–510, Jan. 1993.
- [32] R. Seidel. Linear programming and convex hulls made easy. In *SCG '90: Proceedings of the sixth annual symposium on Computational geometry*, pages 211–215, New York, NY, USA, 1990. ACM Press.
- [33] M. I. Shamos. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, 1978.
- [34] R. Wenger. Randomized quickhull. *Algorithmica*, 17(3):322–329, 1997.