Kinetic Algorithms via Self-Adjusting Computation

Umut A. Acar* Guy E. Blelloch† Kanat Tangwongsan†
Jorge L. Vittes‡

February 22, 2006
CMU-CS-06-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

* Toyata Technological Institute at Chicago. E-mail: umut@tti-c.org
† Computer Science Department, Carnegie Mellon University. E-mail: {blelloch, ktangwon}@cs.cmu.edu
‡ Stanford University. E-mail: jvittes@stanford.edu

Abstract

Define a *static algorithm* to be an algorithm that computes some combinatorial property of its input consisting of static, *i.e.*, non-moving, objects. In this paper, we describe a technique for syntactically transforming static algorithms into *kinetic algorithms*, which compute the statically computed property under motion, à la kinetic data structures. Based on the properties of the transformation technique, we give an algorithm for performing robust motion simulations with fixed-precision floating-point arithmetic. To evaluate the practical effectiveness of the approach, we implement a library for performing the transformation, transform a number of algorithms and give a detailed experimental evaluation. The results show that the technique makes it easy to implement robust kinetic algorithms and delivers good performance in practice.

Keywords: Computational geometry, kinetic data structures, kinetic algorithms, self-adjusting computation, convex hulls

1 Introduction

Since first proposed by Basch, Guibas, and Hershberger [9], many *kinetic data structures* for computing combinatorial properties of moving object have been devised and analyzed (*e.g.*, [5, 8, 6]). Some kinetic data structures have also been implemented [10, 8, 14]. A kinetic data structure for computing a property can be viewed as maintaining the proof obtained by running a static algorithm which computes that property. Based on this connection between static algorithms and kinetic data structures, previous work developed kinetic data structures by *kinetizing* static algorithms. In all previous approaches, the kinetization process is performed manually.

This paper proposes the first technique for kinetizing static algorithms semi-automatically by applying a syntactic transformation, presents techniques for robust kinetic simulations using finite-precision floating-point arithmetic, and evaluates the effectiveness of the approach by considering a number of algorithms. The transformation (Section 2) relies on self-adjusting computation [1], where (self-adjusting) programs can respond to any change to their data (*e.g.*, insertions/deletions into/from the input, changes to the outcomes of comparisons) by running a general-purpose *change-propagation algorithm*. Once transformed into a self-adjusting algorithm, an algorithm for computing a property of static objects can be kinetized by pairing it with a kinetic event scheduler.

An important problem in motion simulation is ensuring robustness in the presence of numerical errors that arise when computing the roots of certain polynomials, called *certificate polynomials*. These roots give the *failure times (events)* at which the computed property may change. We describe a scheduling algorithm that can guarantee robustness even with finite-precision floating-point arithmetic (Section 2.2). The main idea behind our approach is to process the events that are closer than the smallest computable precision together as a batch. In all previous work, events are processed one by one. This requires computing the order of events exactly, by using numerical techniques based on exact and/or interval arithmetic [16, 15, 14]. It is well-known that these techniques can be expensive in practice. The reason for processing events one by one is that events may be interdependent: processing one may invalidate another. Our approach is made possible by the ability of the change-propagation algorithm to process interdependent events correctly. It is not known if previously proposed approaches based on kinetic data structures can be extended to support interdependent events (efficiently).

Our proposed approach also has important software engineering benefits. The approach enables composing kinetized algorithms (sending the output of one algorithm to another as input), and integrating dynamic changes (insertions/deletions) and kinetic changes (changes to the outcomes of certificates) without making any changes to the implementation. In previous work, these require implementing additional data structures (*e.g.*, chapter 9 in Basch's thesis [8]). Guibas identifies both of these as important problems concerning the implementation of kinetic data structures [13].

To evaluate the effectiveness of our approach, we implement a library for kinetizing static algorithms (Section 3) and kinetize a number of algorithms (Section 4) by applying the proposed transformation technique. The kinetized algorithms, which we call *kinetic (self-adjusting) algorithms*, include the merge-sort and the quick-sort algorithms, the Graham-Scan [12], merge-hull, quick-hull [7], ultimate [11] algorithms for computing convex hulls, and Shamos's algorithm for computing diameters [18]. Based on these applications, we perform an extensive experimental evaluation. The experiments show that kinetic algorithms can process various kinds of changes efficiently and robustly. The results show that our implementation has reasonably small constant-factor overheads by comparisons to a synthetic benchmark, and that kinetic algorithms can be as much as three orders of magnitude faster than recomputing from scratch.

2 From Static to Kinetic Programs

We describe the transformation from static to kinetic algorithms, and present an algorithm for robust motion simulation by exploiting certain properties of the transformation (Section 2.2). The asymptotic complexity of kinetic algorithms can be determined by analyzing the *stability* of the program; we describe stability briefly in Section 2.3.

2.1 The Transformation

The transformation of a static program (algorithm) into a kinetic program first requires transforming the static program into a self-adjusting program, and then linking the self-adjusting program with a kinetic scheduler.

Transforming a static program into a self-adjusting program relies on operations for creating, reading, and writing modifiable references and for memoization. A *modifiable reference* or, *modifiable* for short, is a reference, whose contents is *changeable*. Once a self-adjusting program executes, the contents of modifiables can be changed, and the computation can be updated by running a *change-propagation algorithm*. The transformation involves two steps. First, the changeable (time-dependent) data are placed into modifiables. For the purposes of this paper, changeable data consists of all comparisons that involve moving points, and the "next pointers" in the input list. Placing the outcomes of comparisons into modifiables enables changing them as points move; placing the links into modifiables enables inserting/deleting elements into/from the input. Second, the programmer inserts operations for modifiable references and memoization. The transformation is aided by language techniques that ensures correctness [1, 2, 4].

Kinetizing a self-adjusting program requires replacing the comparisons in the program with certificate-generating comparisons. In terms of programming, this is achieved by linking the program with a library that provides certificate-generating comparisons. When executed, a certificate-generating comparison creates a *certificate* consisting of a boolean value and a *certificate function* that represents the value of the certificate over time. When a certificate is created, its *failure time* is computed by finding the roots of its certificate function, and the certificate is inserted into a priority queue, called the *certificate queue* based on its failure times. Using the certificate queue, an *event scheduler* simulates motion by repeatedly removing the earliest certificate to fail, changing the value of the certificate (from `true` to `false` or vice versa) and updating the computation by running change propagation. As examples, Appendix A gives the code for self-adjusting `minimum` and `quick-hull` applications.

The key difference between our approach and the previously proposed approaches is the use of change propagation for adjusting computations to changes. Instead of requiring the design of a separate kinetic data structure, the change-propagation algorithm takes advantage of the problem structure as expressed by the static algorithm to update the output effectively. Since change-propagation is general purpose and can handle any change to the computation, the approach guarantees some properties:

Theorem 1 (Properties of Kinetic (Self-Adjusting) Algorithms) *Kinetic algorithms obtained by self-adjusting-computation transformation satisfy the following properties:*

- **Integrated Changes:** *They can adjust to any change to their data including any combination of changes to the input (*a.k.a.*dynamic changes), and changes to the outcomes of comparisons (*a.k.a.*kinetic changes).*
- **Composibility:** *They are composable: if `f(·)` and `g(·)` are kinetic (self-adjusting) algorithms, then so is `f(g(·))`.*

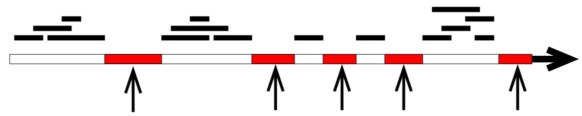

Figure 1: The simulation time, the certificate failure intervals, and some safe times (upward arrows).

- **Time Advancing:** *In a kinetic simulation with a kinetic (self-adjusting) algorithm, the simulation time can be advanced from the current time to any time in the future. This requires first changing the outcome of certificates that fail between the current time and t, and then running change propagation.*

2.2 Robust Motion Simulation

Traditional approaches to motion-simulation based on kinetic data structures rely on computing the exact order in which certificates fail. The reason for this is correctness. Since comparisons can be interdependent, changing the outcome of one certificate can invalidate (delete) another certificate. Thus, if the failure order of comparisons is not determined exactly, then the event scheduler can incorrectly process an event e_1 prematurely, before the event e_2 that invalidates e_1. This can easily lead to an error because of violating invariants maintained by a kinetic data structure. In general, determining the exact order of certificate failure times requires exact arithmetic. Often, however, exact calculations can be avoided by using approaches based on interval arithmetic. Much of the previous work on robust motion simulation focused on techniques for determining the exact order of failure times by using numerical approaches [16, 15, 14].

We propose an algorithm for robust motion simulation that only requires fixed-precision floating-point arithmetic. The algorithm takes advantage of the time-advancing property of kinetic algorithms (Theorem 1) to perform change-propagation only at "safe" points in time at which the outcomes of certificates can be computed precisely.

Definition 2 (Safe Time Point) *Consider a complete kinetic simulation, where an interval that contains the exact failure time of each certificate is computed. Let C be the set of certificates inserted into the event queue throughout the simulation. We say that time t is safe, if t is not contained in the interval of any certificate in C.*

Figure 1 shows a hypothetical example and some safe times.

If the scheduler could determine the safe time points, then it would perform a robust simulation by repeatedly advancing the time to the next safe *target*, *i.e.*, the next safe time point. Since the outcomes of all comparisons can be determined correctly at safe targets, such a simulation is guaranteed to be correct. It is not possible, however, to know what targets are safe online, because this requires knowing all the future certificates. Our algorithm therefore selects a safe target t based on existing certificates and aborts when it finds that t becomes unsafe. To determine if t becomes unsafe, the algorithm checks, during change propagation, whether a certificate whose interval contains t is created. If t becomes unsafe, then change propagation is aborted and the simulation is restarted at the next δ-safe time greater than t (this ensures progress). Since certificate failure times are the roots of the certificate polynomials, the algorithm is likely to abort if the target is too close to a failure time. To minimize the number of aborts, the algorithm chooses targets that are δ-safe, i.e., δ away from a previous interval. More precisely, a time t is *δ-safe* if the intersection of the interval $(t - \delta, t)$ and any existing interval is empty. To be effective, δ must be greater than the width of the largest certificate interval.

2.3 Stability

The asymptotic complexity of change propagation with a kinetic algorithm can be determined by analyzing the *stability* of the kinetic algorithm. Since this paper concerns experimental issues, we give a brief overview of stability here and refer the reader to the first author's thesis for further details [1]. The stability of an algorithm is measured by computing the "edit distance" between the execution traces of the algorithm on different inputs. For a class of computation, called *monotone*, the execution traces can be represented as sets of the instructions executed by the algorithm, and the edit distance can be computed as the symmetric set difference. For example, the stability of the merge sort algorithm under a change to the outcome of one of the comparisons can be determined by computing the symmetric set difference of the set of comparisons performed before and after this change. Elsewhere [1], we prove a stability theorem that states that, under certain conditions, change-propagation takes time proportional to the edit distance between the traces of the algorithm on the inputs before and after the change.

Many existing algorithms are either stable (often in a randomized sense) or can be made stable by making small changes to their design [1]. Section 4 gives a more detailed description of the stability issues in our applications.

3 Implementation

We implemented a library for transforming static algorithms into kinetic. The library consists of primitives for creating certificates, event scheduling, and is based on a library for self-adjusting-computation. The self-adjusting-computation library is described elsewhere [2, 3]. The implementation of the kinetic event scheduler follows the description in Section 2.2; as a priority queue, a binary heap is used. For solving the roots of the polynomials, the library relies on a degree-two solver based on standard floating-point arithmetic. The solver performs standard floating-point arithmetic and makes no further accuracy guarantees. The full code for the implementation and the code for the applications described below is available at `http://ttic.uchicago.edu/~umut/sting`

4 Applications

Using our library for kinetizing static algorithms, we implemented a number of algorithms and kinetized them. The algorithms include an algorithm for finding the minimum key in a list (`minimum`), the `quick-sort` and the `merge-sort` algorithms, several convex hull algorithms including `graham-scan` [12], `quick-hull` [7], `merge-hull`, the (improved) `ultimate` convex-hull algorithm [11], and an algorithm, called `diameter`, for finding the diameter of a set of points [17]. The input to all our algorithms is a list of one or two dimensional points. Each component of a point is a univariate polynomial of time with floating-point coefficients. In the static versions of the algorithms, the polynomials have degree zero; in the kinetic versions, the polynomials can have arbitrary degree depending on the particular motion represented.

To obtain an efficient kinetic algorithm for an application, we first implement a stable, static algorithm for that application and then transform the algorithm into a kinetic algorithm using the techniques described in Section 2.1. The transformation increases the number of lines by about 20% on average. Composibility (Theorem 1) turns out to be important in our applications. For example, the `quick-hull` and `ultimate` use `minimum` to find the point furthest away from a line; `diameter` uses `quick-hull` to compute the convex hull of the points, and `minimum` to find the furthest anti-podal pair; `graham-scan` uses `merge-sort` to sort its input.

Not every algorithm for solving a problem is stable (and thus not every kinetized algorithm is efficient). For example, the straightforward list-traversal algorithm for computing the minimum of a list of keys is not stable. We give a stable algorithm by using a random-sampling technique (this algorithm is described in more detail in Appendix A). The other algorithms require small changes to ensure stability: the `quick-sort`, `quick-hull`, `graham-scan`, and `diameter` algorithms require no changes. The `merge-sort` and `merge-hull` algorithms require randomizing the split phase so that the input list is randomly divided into two sets (by applying random sampling) instead of dividing in the middle. The `ultimate` convex hull algorithm requires randomizing the elimination step so that points are paired randomly by using a random-mate technique.

5 Experimental Results

We present an experimental evaluation of the approach. We give detailed experimental results for the `diameter` application, give a summary of the results for other applications. We finish by comparing the convex-hull algorithms and discussing the effectiveness of our robust scheduling algorithm.

To evaluate the performance of the approach, we report speedups compared to from-scratch execution. The speedup measures show that the approach yields near linear speedups for all applications. In addition, we report experiments with a synthetic benchmark that enables measuring the constant factors involved in our implementation. We also tried to compare our implementation to the implementation of kinetic convex-hulls by Basch et al [10]. Unfortunately, we could not compile their implementation on none of the various systems that are available to us, because the implementation relies on depreciated libraries.

Experimental Setup. We ran our experiments on a 2.7GHz Power Mac G5 with 4 gigabytes of memory. We compiled the applications with the MLton compiler using "`-runtime ram-slop 1`" option that directs the run-time system to use all the available memory available on the system—MLton, however, can allocate a maximum of about two gigabytes. Since MLton uses garbage collection, the total time depends on the particulars of the garbage-collection system, we therefore report the *application time*, measured as the total time minus the time spent for garbage collection (garbage collection is discussed elsewhere [3]). For the experiments, we use a standard floating-point solver with the robust kinetic scheduler (Section 2.2). We assume that certificate failure times are computed within an error of $\pm 10^{-10}$ and choose $\delta = 2 \times 10^{-10}$.

Input Generation. We generate the inputs for our experiments randomly. For one-dimensional applications, we generate points uniformly at random between 0.0 and 1.0 and assign them velocities uniformly at random between -0.5 and 0.5. For two-dimensional applications, we pick points from within the unit square uniformly at random and assigning a constant velocity vector to each point where each component is selected from the interval $[0.5, 0.5]$ uniformly at random.

Measurements. In addition to measuring various quantities such as the number of events in a kinetic simulation, we run some specific experiments to take some specific measurements. For the purposes of determining the constant-factors involved in our approach, we also perform experiments with a synthetic benchmark designed to have the optimal asymptotic complexity for the considered application with a small constant factor. These experiments are described below; throughout, n denotes the input size.

- **Average time for an insertion/deletion:** This is measured by applying a delete-propagate-insert-propagate step to each point in the input. Each step deletes an element, runs change propagation, inserts the element back, and runs change propagation. The average is taken over all propagations. [1]

[1] When measuring these operations, the kinetic event queue operations are turned off.

- **Average time for a kinetic event:** This is measured by running a kinetic simulation and averaging over all events. For all applications except for `graham-scan` and sorting applications, we run the simulations to completion. For sorting and `graham-scan` applications, we run the simulations for the duration of $10 \times n$ events.
- **Average time for an integrated dynamic change & kinetic event:** This is measured by running a kinetic simulation while performing one dynamic change at every kinetic event. Each dynamic change scales the coordinates of a point by 0.8. We run the simulation for the duration of $2 \times n$ events such that all points are scaled twice. The average is taken over all events and changes.
- **Competitiveness-Constant C with respect to a synthetic benchmark:** To determine the effectiveness of change propagation, we perform a kinetic simulation with the following synthetic benchmark based on treaps. The (synthetic) benchmark initializes two treaps A and B to contain all the points in the input. A kinetic simulation is then performed as usual but, at each event, some extra treap operations are performed. In particular, at each event the benchmark selects the next K points from the input and does a search for each point in treap A. For each point p on the search path, K line-side tests (for computational geometry algorithm) or key comparisons (for other algorithms) are performed. In addition, for each test, p is deleted from treap B and inserted back again. We define the *competitiveness constant*, denoted C, for an algorithm to be the value of $C = K^2$ for which the time per event with the synthetic benchmark is at least twice as much as the kinetic time per event (measured without the benchmark).

The synthetic benchmark models a kinetic algorithm: one treap (B) models the kinetic event queue; the other (A) models the kinetic algorithm. The benchmark requires $O(K^2 \log^2 n)$ time for an input with n points. The competitiveness constant $C = K^2$ denotes the constant factor overhead for time per event during a kinetic simulation with respect to a simple $O(\log^2 n)$ benchmark with treaps. Thus, C gives a measure of the constant factors involved in our implementation with respect to a simple algorithm with equivalent asymptotic complexity. We note that the synthetic benchmark could be parametrized with two constants (one for each treap); we chose this model, because it is simpler.

Diameter. The `diameter` application first computes a convex hull of the points, then performs a linear scan of the convex hull to compute the antipodal pairs, and finds the pair that is furthest apart. Our implementation of `diameter` uses `quick-hull` and `minimum` algorithms. Since the computation of the convex-hull is the bottleneck, the performance of `diameter` is very similar to that of `quick-hull`. We note that Agarwal et al give a similar algorithm but provide no implementation [5]. Due to the similarity between computing diameters and width of a point set, we expect a similar technique can be used to compute the width of a point set.

Figure 2 shows the total time for a from-scratch run of the kinetic `diameter` algorithm for varying input sizes. The figure shows that the kinetic algorithm is at most 5 times slower than the static algorithm for the considered inputs—due to the event queue, asymptotic overhead of a kinetic algorithm is $O(\log n)$. Figure 3 shows the total time for complete kinetic simulations of varying input sizes—the curve seems slightly super-linear. Figure 4 shows the average time for change propagation after an insertion/deletion for varying inputs. Figure 5 shows the average time per kinetic event and the average time for an integrated dynamic change and kinetic event. Both curves fit $O(\log^2 n)$. These experimental results match best known asymptotic bounds for the kinetic diameter problem [5].

To get a measure of how fast change propagation is, we compute the average speedup (Figure 6) as the ratio of the average time for one kinetic event to the time for a from-scratch execution of the static version. As can be seen, the speedup increases nearly linearly with the input size to exceed three orders of magnitude.

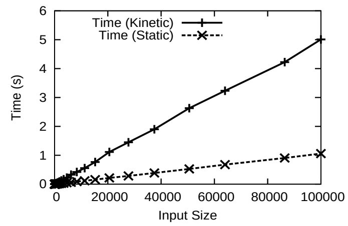

Figure 2: Time (seconds) for initial run versus input size.

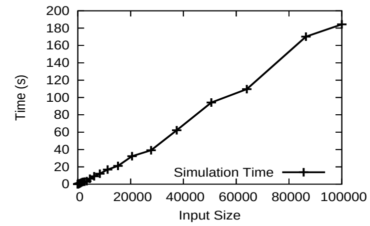

Figure 3: The time (seconds) for a complete kinetic simulation versus input size.

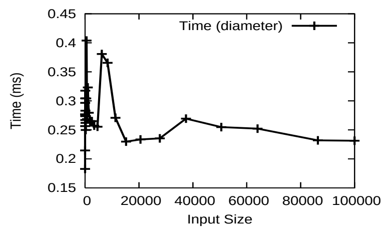

Figure 4: Average time (milliseconds) for an insertion/deletion versus input size.

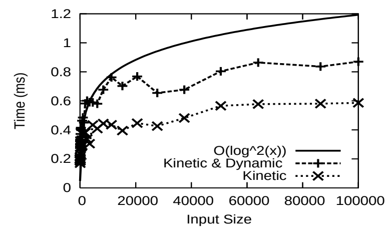

Figure 5: Average time (milliseconds) per kinetic event and integrated kinetic and dynamic events versus input size.

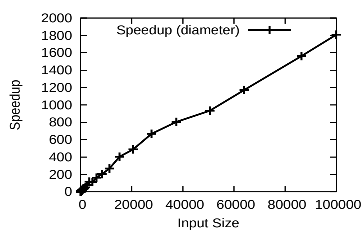

Figure 6: Average speedup of a kinetic event with respect to recomputing from scratch.

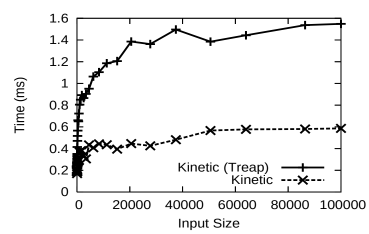

Figure 7: Average time (milliseconds) per kinetic event with `diameter` and with the synthetic benchmark.

Figure 7 shows the average time per event with the synthetic benchmark with $K = 3$ versus time per event with a kinetic simulation. The result shows that the synthetic benchmark is more than twice slower. Thus, the competitiveness factor for `diameter` is equal to $K^2 = 9$.

Other benchmarks. We report a summary of our results for other benchmarks at fixed inputs sizes. Table 1 shows, for input sizes ("n"), the timings for from-scratch executions of the static version ("Static Run") and the kinetic version ("Kinetic Run"), the overhead, the average time for change propagation after an insertion/deletion ("Insert/Delete"), and the speedup of change propagation computed as the average time for an insertion/deletion divided by the time for recomputing from scratch using the static algorithm. The overhead, defined as the ratio of the time for a kinetic run to the time for a static run, is $O(\log n)$ asymptotically because of the certificate-queue operations. The experiments show that the overhead is about 9 on average for the considered inputs, but varies significantly depending on the applications. As can be expected, the more sophisticated the algorithm, the smaller the overhead, because the time taken by the library operations (operations on certificates, event queue, modifiables, etc.) compared to the amount of "real" work performed by the static algorithm is small for more sophisticated algorithms. In terms of the time

Application	n	Static Run	Kinetic Run	Overhead	Insert Delete	Speedup
minimum	10^6	0.8	8.0	10.5	1.6×10^{-5}	> 50000
merge-sort	10^5	1.3	9.7	7.4	3.6×10^{-4}	> 4000
quick-sort	10^5	0.3	9.8	31.6	3.7×10^{-4}	> 800
graham-scan	10^5	2.3	12.5	5.4	8.0×10^{-4}	> 3000
merge-hull	10^5	2.2	10.0	4.7	6.0×10^{-3}	> 300
quick-hull	10^5	1.1	5.0	4.7	2.1×10^{-4}	> 5000
ultimate	10^5	1.8	7.8	4.2	1.0×10^{-3}	> 1500
diameter	10^5	1.1	5.0	4.7	2.3×10^{-4}	> 5000

Table 1: From-scratch runs and dynamic changes.

Application	n	Static Run	Simulation	# Events	# Ext. Events	Per Event	Per Int. Event	Speedup	C
minimum	10^6	0.8	49.7	5.3×10^5	9	9.3×10^{-5}	9.3×10^{-5}	> 8000	1
merge-sort	10^5	1.3	239.1	10^6	10^6	2.4×10^{-4}	9.8×10^{-4}	> 6000	4
quick-sort	10^5	0.3	430.9	10^6	10^6	4.3×10^{-4}	2.9×10^{-2}	> 700	9
graham-scan	10^5	2.3	710.3	10^6	38	7.1×10^{-4}	1.4×10^{-3}	> 3000	9
merge-hull	10^5	2.2	1703.6	6.8×10^5	293	2.5×10^{-3}	7.4×10^{-3}	> 800	16
quick-hull	10^5	1.1	171.9	3.1×10^5	293	5.6×10^{-4}	8.9×10^{-4}	> 2000	9
ultimate	10^5	1.8	1757.8	4.1×10^5	293	4.3×10^{-3}	7.3×10^{-3}	> 400	25
diameter	10^5	1.1	184.4	3.1×10^5	11	5.9×10^{-4}	8.7×10^{-4}	> 2000	9

Table 2: Kinetic simulations with (also with integrated dynamic changes).

for insertions/deletions, both sorting algorithms seem to perform similarly; the convex-hull algorithms can be ranked from best to worst as `quick-hull`, `graham-scan`, `ultimate`, and `merge-hull`; the `diameter` application performs very similarly to `quick-hull`. As the "speedup" column shows change propagation can be orders of magnitude faster than recomputing from scratch.

Table 5 shows the timings for kinetic simulations. The "n" column shows the input size, "Simulation" column shows the time for a kinetic simulation, the "# Events" and "# Ext. Events" columns show the number of events and external events respectively, the "per Event" column shows the average time per kinetic event. The "per int. ev." column shows the average time for an integrated dynamic and kinetic event. The "Speedup" column shows the average speedup computed as the ratio of time for a from-scratch execution of the static version to the average time for an event. The "C" column shows the competitiveness constant for each algorithm. The competitiveness-constant measures show that the change propagation has relatively small constant factor overheads—the average competitiveness ratio (over all algorithms) is about 10. As the speedup column shows, the change propagation is orders of magnitude faster than re-computing from scratch. The average speedup (over all algorithms) more than 2500.

The results show that, of sorting algorithms, `merge-sort` is more effective than `quick-sort`; `merge-sort` is two times faster for kinetic events, and nearly thirty times faster for integrated events. For the convex hull-algorithms, we need a more detailed discussion but it is clear that the `graham-scan` algorithm is not effective, because it requires too many events (as it is sorting based)—note that `graham-scan` simulations are not run up to completion. The results show that the performance properties of the `diameter` and the `quick-hull` algorithms are similar as expected.

A comparison of convex hull algorithms. We compare the `quick-hull`, `ultimate`, and `merge-hull` algorithms based on their *responsiveness, efficiency* and *locality*. These properties help determine the effec-

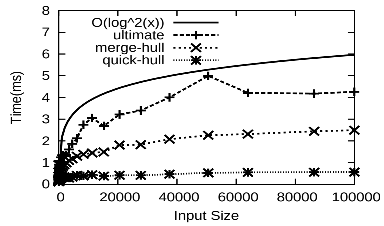

Figure 8: Average time (milliseconds) per kinetic event for some convex-hull algorithms.

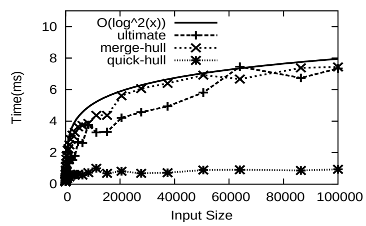

Figure 9: Average time (milliseconds) per integrated kinetic and dynamic events for some convex-hull algorithms.

tiveness of kinetic algorithms [9]. For brevity and because it is not practical, we do not discuss `graham-scan` in detail.

Figure 8 shows the time per event for the convex hull algorithms. The time per event measures the *responsiveness* of a kinetic algorithm. As can be seen, from best to worst responsiveness, the algorithms rank as `quick-hull`, `merge-hull`, and `ultimate`. In a kinetic simulation, the total number of events processed determines the *efficiency* of an algorithm. In terms of efficiency the algorithms rank from best to worst as `quick-hull`, `ultimate`, and `merge-hull` (Table 5). The total time for a simulation gives a measure of the effectiveness of a kinetic algorithm. The `quick-hull` algorithm is the most effective. The `merge-hull` and `ultimate` algorithm differ slightly but are a factor of four slower than `quick-hull`.

Kinetic algorithms can also be compared based on their *locality* [9], which is defined as the maximum number of certificates that depend on any input point. The time for integrated dynamic and kinetic changes (Figure 9) gives a measure of locality because a change to the coordinates of a point requires recomputing all certificates that depend on that point. For `quick-hull` and `ultimate` integrated changes are slightly slower than just kinetic changes. For `merge-hull` integrated changes are about a factor of two slower. In terms of their locality, the algorithms rank from best to worst as `quick-hull`, `ultimate`, and `merge-hull`.

The results show that due to the sorting step, the `graham-scan` algorithm is not suited for kinetic simulations, whereas `quick-hull` appears to perform the best. One disadvantage of `quick-hull` is that it is difficult to prove asymptotic bounds for it—for `merge-hull` and `ultimate` bounds can be given. If asymptotic complexity is important, then the experiments indicate that `merge-hull` algorithm is more slightly more efficient than `ultimate` if no dynamic changes are performed. In the presence of dynamic changes, `ultimate` will likely be more efficient.

Robustness. Our experiments rely on the robust scheduling algorithm described in Section 2.2. To determine the effectiveness of the approach, we performed additional testing by running kinetic simulations and probabilistically verifying the output after each kinetic event. These experiments showed that the approach ensures correctness for all inputs that we considered: up to 100,000 points with all applications.[2] With computational geometry algorithms, the scheduler performed no cold restarts. With sorting (and `graham-scan`) algorithms, there were ten restarts with 100,000 points—no restarts took place for smaller inputs. Since sorting algorithms can process up to $O(n^2)$, this is not surprising.

As described in Section 2.2, the robust scheduling algorithm can process multiple certificates at once to ensure correctness. We measured the number of certificates processed at each event to be less than 1.75 averaged over all our applications. The quantity increases quadratically with input size for sorting based

[2]These limits are due to memory limitations of the MLton compiler. We could run some applications with more than 300,000 points.

applications (the maximum is 3.0 with `graham-scan`), but grows very slowly for other applications. We note that both the number of restarts and the number of certificates can be further decreased by increasing the precision of root computations.

6 Conclusion

This paper describes the first technique for kinetizing static algorithms by applying a syntactic transformation and gives a scheduling algorithm for robust kinetic simulations using fixed-precision floating-point arithmetic. The effectiveness of the technique is evaluated by applying it to a number of algorithms and performing an extensive experimental evaluation. The approach makes it possible to integrate dynamic and kinetic changes, and compose kinetic algorithms without making any changes to their implementation. The experimental results show that the approach enables correct simulations and performs well in practice.

References

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

[2] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. In *ACM SIGPLAN Workshop on ML*, 2005.

[3] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation, 2006. To Appear in the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.

[4] 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.

[5] Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika Rauch Henzinger. Parametric and kinetic minimum spanning trees. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 596–605, 1998.

[6] Pankaj K. Agarwal, Leonidas J. Guibas, John Hershberger, and Eric Veach. Maintaining the extent of a moving set of points. *Discrete and Computational Geometry*, 26(3):353–374, 2001.

[7] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[8] Julien Basch. *Kinetic Data Structures*. PhD thesis, Department of Computer Science, Stanford University, June 1999.

[9] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.

[10] Julien Basch, Leonidas J. Guibas, Craig D. Silverstein, and Li Zhang. A practical evaluation of kinetic data structures. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 388–390, New York, NY, USA, 1997. ACM Press.

[11] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996.

[12] R. L. Graham. An efficient algorithm for determining the convex hull of a finete planar set. *Information Processing Letters*, 1:132–133, 1972.

[13] L. J. Guibas. Kinetic data structures—a state of the art report. In *Proceedings of the Third Workshop on Algorithmic Foundations of Robotics*, 1998.

[14] Leonidas Guibas, Menelaos Karaveles, and Daniel Russel. A computational framework for handling motion. In *Proceedings of teh Sixth Workshop on Algorithm Engineering and Experiments*, pages 129–141, 2004.

[15] Leonidas Guibas and Daniel Russel. An empirical comparison of techniques for updating delaunay triangulations. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 170–179, New York, NY, USA, 2004. ACM Press.

[16] Leonidas J. Guibas and Menelaos I. Karavelas. Interval methods for kinetic simulations. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 255–264. ACM Press, 1999.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag Inc., 1985.

[18] Michael I. Shamos. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, 1978.

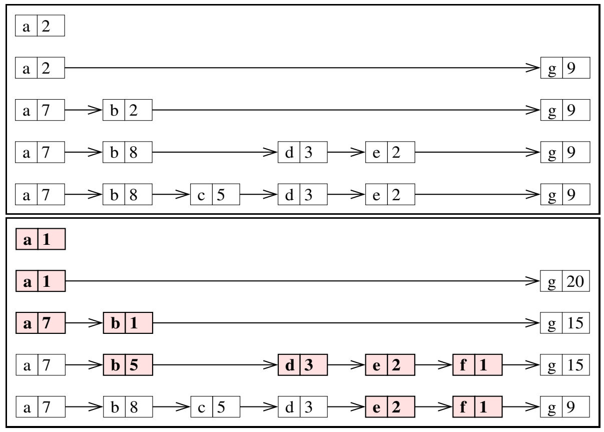

Figure 10: Minimum finding before and after inserting 1.

A Transforming a static program into a self-adjusting program

Static programs can be transformed into self-adjusting programs by means of a syntactic transformation. The resulting self-adjusting programs are trivially kinetized by linking them with certificate-generating comparisons and an event scheduler. The performance of change propagation under updates to the data of a self-adjusting programs depends on the stability of the transformed static algorithm. This section gives some detail about the transformation and considers an example where a static algorithm is not stable. A more precise treatment of stability and the transformations, and more examples can be found in Acar's thesis [1].

Giving an efficient kinetic algorithm to a problem involves first coming up with a stable algorithm and then transforming that algorithm into a self-adjusting program. Many algorithms are already stable or can be made stable by small changes. For example, an efficient algorithm for kinetic convex hulls can be given by kinetizing the quick-hull algorithm. Although this algorithm is difficult to prove stable (as it is difficult to prove efficient in the static case), experiments confirm that it is stable in practice (Section 5). The code for self-adjusting/kinetic convex-hull is given in Figure 12. The transformation is described in more detail below.

Sometimes, a straightforward algorithm that is efficient for the static problem is not stable. As an example, consider the problem of finding the minimum key in a list. A straightforward solution is the linear-scan algorithm that scans the list from the beginning to the end while maintaining the minimum of keys visited so far. This algorithm is not stable, because it can compare the current minimum to many keys—changing the minimum can require performing many comparisons. This problem can be solved by a stable algorithm by using random sampling. The idea is to delete a random subset of the keys after incorporating their minimum to their live neighbors. If, for example, each key is selected with probability say 0.5, then one application of this select-and-contract step reduces the length of the list by 0.5 in expectation. It can be shown that after an expected logarithmic number of steps, the list is reduced to a single key that holds the minimum of all keys. We show elsewhere that this algorithm is expected $O(\log n)$-stable under insertions/deletions [1]. Using the stability theorem, this bounds yields an expected-$O(\log n)$ bound on change propagation.

Figure 10 shows the traces for the executions of the algorithm on inputs that differ by the key **1**. The figure highlights the data that differs between two traces—the change propagation algorithm will compute the highlighted data after inserting the key **1**. The ordering of points can also be changed by changing the outcomes of the comparisons performed in the execution. For example, the relative ordering of the keys 2 and 3 can be swapped. Running change propagation after such a change will propagate the new minimum (which is now 3) through the computation.

The Transformation. The transformation of a static program into a self-adjusting program relies on several operations for supporting modifiable references and memoization. A *modifiable reference* or, *modifiable* for short, is a reference that holds a *changeable* value. Modifiable references are created by the `mod` operation, read by the `read` operation, and written by the `write` operation. In addition to modifiable references, we assume a `lift` operation for memoizing function calls.

The transformation involves two steps. First the programmer places the computation data that is expected to change over time (changeable data) into modifiables. Second, the program is updated by inserting, `mod` `read`, `write`, `lift` operations. The transformation can be greatly aided by a type system that ensures that the operations are placed appropriately and safely. In other work, we describe type systems and language techniques required to ensure safety [4, 2].

As an example, Figure 11 shows an implementation of the stable, static algorithm for computing the minimum described in Section 2, and its self-adjusting/kinetic version. The programs are written in the ML language (some slight sugaring is used to increase clarity). The pieces of code that needs to be inserted to the static algorithm are underlined. For brevity, we use $\Box\!\!\rightarrow$ to denote a `read` operation. The static program takes as input a comparison function `comp` of type α `*` α $\rightarrow$ `bool` and a list `l` of type α `list` and returns the minimum element in the list with respect to `comp`. To transform the static program into the self-adjusting program, the programmer changes the input type from an ordinary list into a *modifiable lists*. A modifiable list is similar to an ordinary lists except that each "tail" is placed in a modifiable. This is similar to the representation of linked lists in imperative languages such as C/C++. The programmer then changes the type of the comparison function so that it returns a modifiable that contains the outcome instead of just the outcome. This can be seen from the specified type of the `minimum` function. This completes the first step of the transformation. The second step involves inserting `mod`, `read`, `write`, and `lift` operations. This step is guided by the restriction that the value of a modifiable reference can only be accessed via a `read`, and a `read` can only return a value through a `write`. A `write` must take place within the context of a `mod`. These restriction enforce the notion that if a value if changeable, then any values computed based on the values are also changeable. Finally certain functions are memoized by wrapping them with a `lift` function. The transformation is aided by a type system that ensures that the operations are placed appropriately and safely [4, 2].

Placing the tail into modifiables enables inserting/deleting elements into/from the list after an execution completes. Placing the comparisons into modifiables enables changing the outcomes of the comparisons; this in turn enables simulating motion via change propagation.

Figure 12 shows the self-adjusting/kinetic version for the quick-hull algorithm obtained by applying the transformation to the static quick-hull algorithm. The code relies on some primitives on modifiable lists such as `filter`, and a module that supplies the geometric primitives specified by the `POINT` signature. Note that the outcome of the comparisons specified by the `POINT` signature all have return type `mod`. This indicates that their values can change over time. These operations generate the certificates. The algorithm relies on the `minimum` function (Figure 11) for finding the furthest point away from the current split line. This use is enabled by the composibility of self-adjusting/kinetic algorithms.

```
type α list = nil
            | cons of α * α list

minimum ::  (α * α → bool) → α list → α

fun minimum comp l =
let
  fun halfList l =
  let
    hash = Hash.new ()
    fun run(v,l) =
      case l of
        nil ⇒ (v, nil)
      | cons(h,t) ⇒
        let min =
          if comp(h,v) then h
          else v
        in if (hash h = 0) then
             (min,t)
           else
             run(min,t))
        end

    fun half l =
      case l of
        nil ⇒ nil
      | cons(h,t) ⇒
          let (v,t') = run (h,t)
          in
            cons(v, half t')
          end
  in half l end

  fun comb l =
    if (length l < 2) then
      case l of nil ⇒ raise Empty
              | cons(h,_) ⇒ h)
    else comb (halfList l))
in comb l end
```

```
type α modlist = NIL
               | CONS of α * (α modlist mod)

minimum ::  (α * α → bool mod ) → α modlist → α

fun minimum comp l =
let
  fun halfList l =
  let
    hash = Hash.new ()
    fun run(v,l) = l □→ c
      case c of
        NIL ⇒ write (v, NIL)
      | CONS(h,t) ⇒ comp (h,v) □→ b
        let min =
          if b then h
          else v
        in if (hash h = 0) then
             (min,t)
           else
             run(min,t))
        end

    fun half c =
      case c of
        NIL ⇒ NIL
      | CONS(h,t) ⇒ lift (h,t) (fn m ⇒
          let p = mod (m □→ (fn t ⇒ run (h,t))
          in p □→ (fn (v,t') ⇒
               cons(v, half t'))
          end

  in l □→ (fn c ⇒ half c) end

  fun comb l = mod ((length l) □→ (fn b ⇒
    if b then
      case l of NIL ⇒ raise Empty
              | CONS(h,_) ⇒ write h)
    else comb (halfList l)))
in comb l end
```

Figure 11: The static algorithm for minimum finding and its kinetic version.

```
signature POINT =
sig
  type t
  val aboveLine :  (t * t) → t → bool mod
  val furthest :  (t * t) → (t *t) → bool mod
  val minX : (t * t) → bool mod
  val maxX : (t * t) → bool mod
end
structure P: POINT = struct ...  end

structure QuickHull =
struct
  fun split (rp1, rp2, ps, hull) =
        fun splitM (p1, p2, ps, hull) =
          let val l = filter (P.aboveLine (p1,p2)) ps
          in l □→ (fn cl ⇒
              case cl of
                NIL ⇒ write (CONS(p1,hull))
              | CONS(h,t) ⇒
                  let val rmax = minimum (P.furthest (p1,p2)) l
                      val rest = mod (rmax □→ (fn max ⇒ splitM (max,p2,l,hull)))
                  in rmax □→ (fn max ⇒ splitM (p1,max,l,rest)) end))
          end
    in rp1 □→ (fn p1 ⇒ rp2 □→ (fn p2 ⇒ splitM (p1,p2,ps,hull))) end

  fun qhull l =
    mod ((length< (l, 2)) □→ (fn b ⇒
      if b then write NIL
      else let val min = minimum (select P.minX) l
               val max = minimum (select P.maxX) l
               val lowerHull = mod (split(max,min,l, mod (write NIL)))
            in split (min, max, l, lowerHull) end))
end
```

Figure 12: Self-Adjusting/Kinetic Quick Hull. Changes to the static algorithm are underlined.