Kinetic Algorithms via Self-Adjusting Computation

Umut A. Acar[1], Guy E. Blelloch[2], Kanat Tangwongsan[2], and Jorge L. Vittes[3]

[1] Toyota Technological Institute
[2] Carnegie Mellon University
[3] Stanford University

Abstract. Define a *static algorithm* as 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 properties of moving objects. The technique offers capabilities for composing kinetic algorithms, for integrating dynamic and kinetic changes, and for ensuring robustness even with fixed-precision floating-point arithmetic. To evaluate the effectiveness of the approach, we implement a library for performing the transformation, transform a number of algorithms, and give an experimental evaluation. The results show that the technique performs well in practice.

1 Introduction

Since first proposed by Basch, Guibas, and Hershberger [13], many *kinetic data structures* for computing properties of moving objects have been designed and analyzed (e.g., [8, 12, 9]). Some kinetic data structures have also been implemented [14, 12, 17]. A kinetic data structure for computing a property can be viewed as maintaining the proof obtained by running a static algorithm for computing 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 techniques for kinetizing static algorithms semi-automatically by applying a syntactic transformation. We call such kinetized algorithms as *kinetic algorithms*. The transformation (Section 2) relies on self-adjusting computation [1], where 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*. We evaluate the effectiveness of the approach by kinetizing a number of algorithms (Sections 3 and 4), including the merge-sort and the quick-sort algorithms, the Graham-Scan [16], merge-hull, quick-hull [11], ultimate [15] algorithms for computing convex hulls, and Shamos's algorithm for computing diameters [21] and performing an experimental evaluation. Our experiments (Section 5) show that kinetized algorithms are efficient in practice.

For the transformation to yield an efficient kinetic algorithm, the static algorithms being transformed need to be *stable*. Informally, an algorithm is stable if a small change to the input causes a small change in the execution of the algorithm. In previous work [1, 6] we have formalized the notion of stability and described approaches to analyzing it.

In practice many algorithms seem stable with respect to small input changes, or can be made stable with minor modifications. From a theoretical point of view, our approach can be viewed as a reduction from dynamic/kinetic problems to stable, static problems. Given that the algorithm designer needs to analyze the stability, one may wonder what advantages the approach has over direct design of kinetic data structures, which often also start by considering static algorithms.

We briefly describe here several advantages of the approach over traditional approaches. In addition to guaranteeing the correctness of kinetized algorithms, the approach enables some capabilities that can dramatically simplify the design and implementation of algorithms for motion simulation. These capabilities are inherent to the approach (require no changes to the implementation) and include composibility, integration of dynamic and kinetic changes, and the ability to advance the simulation time to any time in the future. *Composibility* refers to the ability to send (or pipe) the output of one kinetic algorithm to another: e.g., if `f(·)` and `g(·)` are kinetic algorithms, then `f(g(·))` is a kinetic algorithm. Composibility is important for building large software systems from smaller modules. *Integration of dynamic and kinetic changes* refers to the ability of kinetic algorithms to respond to both dynamic changes (e.g., insertions/deletions into/from the input), and kinetic changes due to motion. With previously proposed approaches, integrating dynamic and kinetic changes can involve major changes to a dynamic or kinetic data structure. For example, Basch et al.'s kinetic convex-hull data structure [13], which does not handle dynamic changes, is very different from Alexandron et al.'s data structure [10], which supports integrated changes. *Advancing-time* capability refers to the ability to advance the simulation time to any time in the future. In addition to combining time-stepping and kinetic simulation approaches, this capability also helps ensure robustness in the presence of certain numerical inaccuracies (discussed in more detail below). Since the approach makes it possible to transform static algorithms into kinetic semi-automatically and guarantees correctness, it also has some software engineering benefits: only the static code needs to be maintained, documented, and debugged.

An important problem in motion simulation is ensuring robustness in the presence of numerical errors in computing the roots of certain polynomials, called *certificate polynomials*. These roots give the *failure times (events)* at which the computed property may change. Based on the advancing-time capability of kinetic algorithm, we describe a scheduling algorithm that guarantees robustness even with finite-precision floating-point arithmetic (Section 2.2). The idea behind our approach is to process the events closer than the smallest computable precision together as a batch. In all previous work, events are processed one by one by computing their order exactly—this requires expensive numerical techniques based on exact and/or interval arithmetic [19, 18, 17]. 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).

To illustrate the differences between our proposal and traditional approaches based on kinetic data structures, consider the example of computing the convex hull of a set

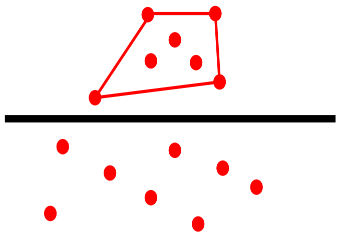

Fig. 1: Computing the convex hull of a set of points that are above some line.

of points above some dividing line (e.g., Figure 1). In the static setting, where points do not move, this can be performed by composing a function `filter_s` that finds the points above a line with the function `hull_s` that computes the convex hull, i.e., the algorithm can be expressed as `hull_s(filter_s(P))`, where `P` is a set of points. Our approach enables giving the kinetic algorithm `hull_k(filter_k(P_k))`, where `hull_k` and `filter_k` are the kinetic versions of `filter_s` and `hull_s` obtained by applying our syntactic transformation, and `P_k` is a set of moving points. This algorithm supports the aforementioned capabilities without further modifications to the implementation.

Suppose we want to solve the same problem by composing the kinetic data structures `filter_kds` and `hull_kds` for the filtering and the convex hull problems. Note first that `hull_kds` must respond to integrated dynamic and kinetic changes, because the output of `filter_kds` will change over time as points cross the dividing line. To compose the two data structures, it is also necessary to convert the changes in the output of `filter_kds` into appropriate insert/delete operations for `hull_kds`. This requires 1) computing the "edit" (changes) between successive outputs of `filter_kds`, 2) implementing a data structure for communicating the changes to `hull_kds` (chapter 9 in Basch's thesis [12]). We don't know of any previously proposed general-purpose approaches to computing "edits" between arbitrary data structures efficiently. Finally, kinetic data structures rely on processing events one by one. This requires sequentializing simultaneous events (e.g., when multiple points cross the dividing line at the same time) and using costly numerical techniques to determine the exact order of failing certificates.

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 requires two steps. First, the static program is transformed into a self-adjusting program. Second, the self-adjusting program is kinetized by linking it with a kinetic scheduler.

Transforming a static program into a self-adjusting program requires annotating the program with primitives for creating, reading, and writing modifiable references, and for memoization. A *modifiable (reference)* is a reference, whose contents is a *changeable* or *time dependent* value. In particular, 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*. 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. After the programmer determines what data is changeable, s/he can transform the program by annotating it with the aforementioned primitives. This transformation is aided by language techniques that ensures correctness [1, 2, 4]. Example transformations can be found elsewhere [7, 1].

Kinetizing a self-adjusting program requires replacing the comparisons in the program with certificate-generating comparisons. This is achieved by linking the program with a library that provides the 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. Creating a certificate requires computing its *failure time* by finding the roots of its certificate function, and inserting the certificate into a *certificate (priority) queue*. An *event scheduler* simulates motion by repeatedly removing the earliest certificate to fail from the certificate queue, changing its outcome, and running the change propagation.

The key difference between our approach and the previously proposed approaches to motion simulation is the use of the change-propagation algorithm for updating computation. Instead of requiring the design of a kinetic data structure, the change-propagation algorithm takes advantage of the computation structure expressed by the static algorithm to update the output. To achieve efficiency, the change-propagation algorithm [3, 1] relies on an integral combination of memoization [5] and dynamic-dependence graphs [6, 4]. Since change-propagation is general purpose and can handle any change to the computation, kinetic (self-adjusting) algorithm have the following capabilities:

- **Integrated Changes:** They can respond 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 algorithms, then so is `f(g(·))`.
- **Advancing Time:** In a kinetic simulation with a kinetic (self-adjusting) algorithm, the simulation time can be advanced from the current time to any time t 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

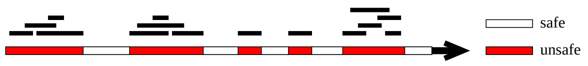

Fig. 2: The simulation time, the certificate failure intervals, and safe and unsafe time intervals.

invalidate (delete) another certificate. Thus, if the failure order of comparisons is not determined exactly, then the event scheduler can prematurely process an event e_1, before the event e_2 that invalidates e_1. This can easily lead to an error by violating critical invariants. Previous work on robust motion simulation focused on techniques for determining the exact order of failure times by using numerical approaches [19, 18, 17].

We propose an algorithm for robust motion simulation that only requires fixed-precision floating-point arithmetic. The algorithm takes advantage of the advancing-time property of kinetic algorithms to perform change-propagation only at "safe" points in time at which the outcomes of certificates can be computed precisely. Given a kinetic simulation, where each certificate is associated with an interval that contains its exact failure time, we say that a time t is *safe* if t is not contained in the interval of any certificate. Figure 2 shows a hypothetical example and some safe time intervals.

If the scheduler could determine the safe time points, then it would perform a robust simulation by repeatedly advancing the time to the earliest next safe time, i.e., *target*. 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, which happens if, during the change propagation, a certificate whose interval contains t is created. To abort, the algorithm restarts the simulation at the next safe time greater than t (this ensures progress).

As discussed in Section 5, this approach seems very effective in practice. To ensure robustness, the scheduler needs to process less than two certificates per event (on average), and requires very few restarts.

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 executions of the algorithm on different data as the symmetric set difference of the executed instructions. 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 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.

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 [3, 2]. 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, which uses the standard floating-point arithmetic and makes no further accuracy guarantees. The solver can be extended to solve higher-degree polynomials. The full code for the implementation 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` [16], `quick-hull` [11], `merge-hull`, the (improved) `ultimate` convex-hull algorithm [15], and an algorithm, called `diameter`, for finding the diameter of a set of points [20]. 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 an arbitrary degree depending on the particular motion represented. For our experiments, we only consider linear motion plans; the polynomials therefore have no more than degree 2.

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. Our implementations rely on the capability to compose kinetic algorithms: the `quick-hull` and `ultimate` algorithms use `minimum` to find the point furthest away from a line; `graham-scan` uses `merge-sort` to sort its input points; `diameter` uses `quick-hull` to compute the convex hull of the points and `minimum` to find the furthest antipodal pair, etc.

Not every algorithm is stable. For example, the straightforward list-traversal algorithm for computing the minimum of a list of keys is not stable. Our algorithm for computing the minimum relies on random-sampling (details can be found elsewhere [7, 1]). The other algorithms require small changes to ensure stability: the `merge-sort` and `merge-hull` algorithms require randomizing the split phase so that the input list is randomly divided into two sets (instead of dividing in the middle); the `ultimate` convex-hull algorithm requires randomizing the elimination step; the `quick-sort`, `quick-hull`, `graham-scan`, and `diameter` algorithms require no changes.

5 Experimental Results

We present an experimental evaluation of the approach. We give detailed experimental results for the `diameter` application, and summarize the results for other applications.

We compare our kinetic `minimum` algorithm to a (hand-designed) kinetic data structure for maintaining the minimum [13, 12]. [4] We finish by comparing the convex-hull algorithms and discussing the effectiveness of our robust scheduling algorithm.

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 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 $\pm 10^{-10}$.

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. These experiments are described below; throughout, n denotes the input size (e.g., number of points).

- **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. [5]
- **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.

[4] We also tried to compare our implementation to the implementation of kinetic convex-hulls by Basch et al. [14]. Unfortunately, we could not compile their implementation, because it relies on depreciated libraries.

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

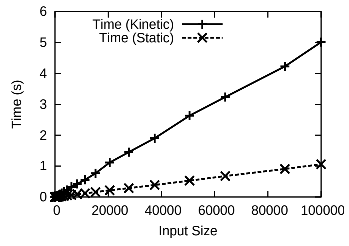

Fig. 3: Time (seconds) for initial run (`diameter`).

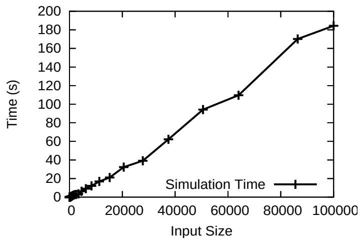

Fig. 4: The time (seconds) for a complete kinetic simulation (`diameter`).

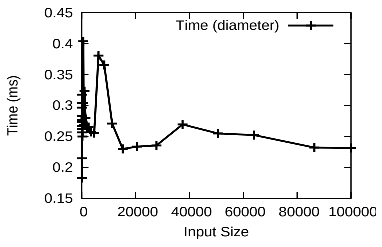

Fig. 5: Average time (milliseconds) for an insertion/deletion (`diameter`).

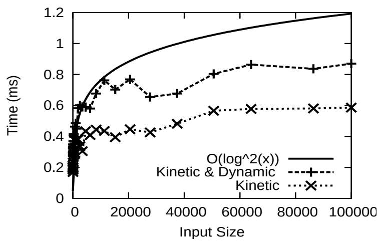

Fig. 6: Average time (milliseconds) per kinetic event and integrated changes (`diameter`).

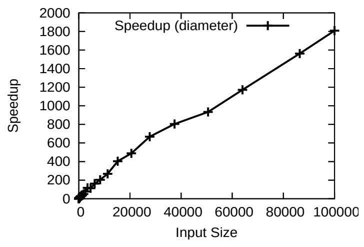

Fig. 7: Average speedup for kinetic events (`diameter`).

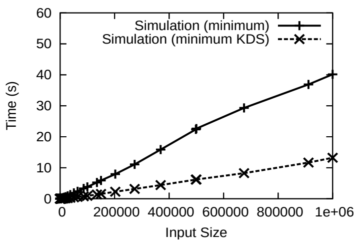

Fig. 8: Simulation time with `minimum` and a tournament-based kinetic data structure.

Diameter. The `diameter` application first computes (using `quick-hull`) the convex hull of its input, performs a scan of the convex hull to compute the antipodal pairs, and finds (using `minimum`) the pair that is furthest apart. We note that Agarwal et al. give a similar algorithm for computing diameters, but they provide no implementation [8]. We expect a similar technique can be used to compute the width of a point set.

Figure 3 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 4 shows the total time for complete kinetic simulations of varying input sizes—the curve seems slightly super-linear. Figure 5 shows the average time for change propagation after an insertion/deletion for varying inputs. Note that the time for change propagation decreases slightly as the input size increases. We believe that this is because the running time

Appli-cation	n	Static Run	Kinetic Run	Over-head	Insert Delete	Speedup
minimum	10^6	0.8	6.2	7.8	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.

Appli-cation	n	Static Run	Simu-lation	# Events	# Ext. Events	Per Event	Per Int. Event	Speedup
minimum	10^6	0.8	40.2	5.3×10^5	9	9.3×10^{-5}	9.3×10^{-5}	> 8000
merge-sort	10^5	1.3	239.1	10^6	10^6	2.4×10^{-4}	9.8×10^{-4}	> 6000
quick-sort	10^5	0.3	430.9	10^6	10^6	4.3×10^{-4}	2.9×10^{-2}	> 700
graham-scan	10^5	2.3	710.3	10^6	38	7.1×10^{-4}	1.4×10^{-3}	> 3000
merge-hull	10^5	2.2	1703.6	6.8×10^5	293	2.5×10^{-3}	7.4×10^{-3}	> 800
quick-hull	10^5	1.1	171.9	3.1×10^5	293	5.6×10^{-4}	8.9×10^{-4}	> 2000
ultimate	10^5	1.8	1757.8	4.1×10^5	293	4.3×10^{-3}	7.3×10^{-3}	> 400
diameter	10^5	1.1	184.4	3.1×10^5	11	5.9×10^{-4}	8.7×10^{-4}	> 2000

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

for `diameter` (and thus change propagation) is sensitive to the size of the convex-hull of the points. In particular, 1) deleting/inserting a point from/into the inside of a convex hull of the input points is cheap and 2) with uniformly randomly distributed points, many of the points are expected to be inside the hull. Figure 6 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 [8]. To measure of how fast change propagation is, we compute the average speedup (Figure 7) 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.

Other benchmarks. We report a summary of our results for other benchmarks at fixed input 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. The "speedup" column shows that 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 speedup column shows that the change propagation is orders of magnitude faster than re-computing from scratch.

The results show that `merge-sort` is more effective than `quick-sort`: the `merge-sort` algorithm is two times faster for kinetic events and nearly thirty times faster for integrated events. We discuss the convex-hull algorithms below.

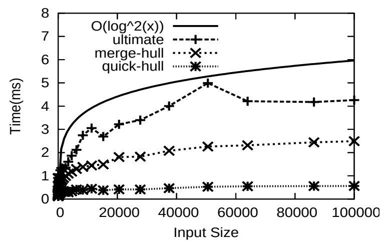

Fig. 9: Average time (milliseconds) per kinetic event for some convex-hull algorithms.

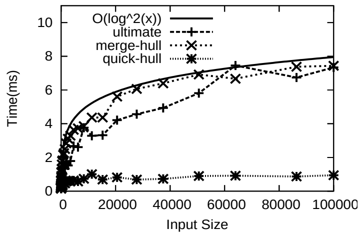

Fig. 10: Average time (milliseconds) per integrated kinetic and dynamic events for some convex-hull algorithms.

A comparison of convex hull algorithms. We compare the `quick-hull`, `ultimate`, and `merge-hull` algorithms based on their *responsiveness, efficiency* and *locality*. which measure the effectiveness of kinetic algorithms [13]. Since `graham-scan` algorithm relies on sorting, it is not practical; we therefore do not discuss `graham-scan` in detail.

Figure 9 shows the time per event for the convex hull algorithms. In a kinetic simulation, the time per event measures the *responsiveness* of a kinetic algorithm, and the total number of events processed determines the *efficiency* of an algorithm. The total simulation time measures overall effectiveness of a kinetic algorithm. In all these respects, the algorithms rank from best to worst as `quick-hull`, `merge-hull`, and `ultimate`.

Kinetic algorithms can also be compared based on their *locality* [13], which is defined as the maximum number of certificates that depend on any input point. The time for integrated dynamic and kinetic changes (Figure 10) gives a measure of locality because a change to the coordinates of a point requires recomputing all certificates that depend on that point. In terms of their locality, the algorithms rank from best to worst as `quick-hull`, `ultimate`, and `merge-hull`.

The results show that the `quick-hull` performs best. One disadvantage of `quick-hull` is that it is difficult to prove asymptotic bounds for it. If asymptotic complexity is important, then the experiments indicate that `merge-hull` algorithm performs better than `ultimate`, especially if few dynamic changes are performed.

Comparison to a handcrafted kinetic data structure. One may wonder how the approach performs relative to handcrafted kinetic data structures. We compare our `minimum` algorithm to the tournament-tree based kinetic data structure for maintaining minimum by Basch, Guibas, and Hershberger [13, 12]. Figure 8 shows the total time of a kinetic simulation with our semi-automatically generated algorithm and the Basch-Guibas-Hershberger kinetic data structure. Our algorithm is a factor of 3 slower than the handcrafted data structure.

Robustness. Our experiments rely on the robust scheduling algorithm (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 tests verified that the approach ensures correctness for all inputs that we considered: up to 100,000 points with all applications.[6] 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.

The robust scheduling algorithm can process multiple certificates simultaneously. We measured the number of certificates processed simultaneously to be less than 1.75 averaged over all our applications. Note that both the number of restarts and the number of certificates can be further decreased by using higher (but still fixed) precision floating-point numbers.

6 Conclusion

This paper describes the first technique for kinetizing static algorithms by applying a syntactic transformation based on self-adjusting computation. The technique ensures that kinetized algorithms are correct, and enables 1) integrating dynamic and kinetic changes, 2) composing kinetic algorithms, and 3) robust motion simulations. The effectiveness of the technique is evaluated by considering a number of algorithms and performing a broad range of experiments. The experimental results show that the approach performs well in practice.

References

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

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

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. 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. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.
6. Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.
7. Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge Vittes. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.
8. 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.
9. 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.
10. Giora Alexandron, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. In *9th Workshop on Algorithms and Data Structures (WADS). Lecture Notes in Computer Science*, volume 3608, pages 269—281, aug 2005.
11. C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.
12. Julien Basch. *Kinetic Data Structures*. PhD thesis, Department of Computer Science, Stanford University, June 1999.
13. Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.
14. 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.
15. Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996.
16. R. L. Graham. An efficient algorithm for determining the convex hull of a finete planar set. *Information Processing Letters*, 1:132–133, 1972.
17. 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.
18. 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.
19. 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.
20. F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag Inc., 1985.
21. Michael I. Shamos. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, 1978.