Dynamizing Static Algorithms, with Applications to Dynamic Trees and History Independence *

Umut A. Acar Guy E. Blelloch Robert Harper Jorge L. Vittes Shan Leung Maverick Woo

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213
{umut,blelloch,rwh,jvittes,maverick}@cs.cmu.edu

Abstract

We describe a machine model for automatically dynamizing static algorithms and apply it to history-independent data structures. Static programs expressed in this model are dynamized automatically by keeping track of dependences between code and data in the form of a dynamic dependence graph. To study the performance of such automatically dynamized algorithms we present an analysis technique based on trace stability. As an example of the use of the model, we dynamize the Parallel Tree Contraction Algorithm of Miller and Reif to obtain a history-independent data structure for the dynamic trees problem of Sleator and Tarjan.

1 Introduction

We describe techniques for automatically generating dynamic algorithms from static ones, and the application of these techniques to history-independent data structures [16, 20, 13]. Our approach is based on having a system track dependences as the algorithm executes. When the input is changed, the system propagates changes through the dependence graph rerunning code where necessary. Since the dependences can change during propagation, the dependence graph is itself dynamic.

To allow the system to track both control and data dependences, the static algorithm must be written for a variant of a standard RAM model. This model is an extension of a model we have previously considered that was based on purely functional programs [1]. To analyze the performance of automatically dynamized algorithm, we use a notion of trace stability based on traces and a distance between traces. We define a *trace* $A_T(I)$ for an algorithm A on input I and a *distance* metric $\delta(\cdot,\cdot)$ between traces. The trace captures, in a particular way, the operations executed by the algorithm. We say that an algorithm is $O(f(n))$-*stable* for a *class of input changes* (*e.g.*, insertions of edges in a forest of size n) if $\max_{(I,I')\in\Delta_n} \delta(A_T(I), A_T(I')) \in O(f(n))$, where Δ_n is the relation mapping inputs to changed inputs for instances of size n. We then prove that if an algorithm is $O(f(n))$-stable, then the input changes can be automatically propagated in $O(f(n)\log f(n))$ time. In a certain special case we show how this can be improved to $O(f(n))$ time.

We apply our approach to the problem of maintaining dynamic trees [25]. We show that a simple variant of the random-mate parallel tree contraction algorithm of Miller and Reif [17] is expected $O(\log n)$-stable for edge insertions and deletions. Since the algorithm falls into the special case mentioned above, it can be automatically dynamized to support updates in expected $O(\log n)$ time. The algorithm uses $O(\log^2 n)$ random bits, and the expectation is over the choices of these bits. The basic algorithms uses $O(n \log n)$ space, but we describe a variant which uses $O(n)$ space. The algorithm is significantly simpler than current $O(\log n)$ worst-case algorithms [25, 12, 15]. In related work [2], we implemented the data structure, which we call RC-Trees (Rake-and-Compress Trees), and applied it to a broad set of applications.

Our automatic dynamization technique builds a dynamic dependence graph to represent an execution and and uses a change propagation algorithm to update the output and the dependences whenever the input changes [1]. A dynamic dependence graph maintains the relationship between code and data in a way that makes it easy for the change-propagation algorithm to identify code, called a *reader*, that depends on a changed value. Readers are stored as closures, *i.e.*, functions with environments. This enables re-execution of a reader in the same state as before—modulo of course the changed values. The change-propagation algorithm maintains a queue of readers affected by changes and re-executes them in sequential execution order. When a reader is re-executed, the dependences

*This work was supported in part by the National Science Foundation under the grant CCR-9706572 and also through the Aladdin Center (www.aladdin.cs.cmu.edu) under grants CCR-0085982 and CCR-0122581.

that it created in the previous execution are deleted and dependences created during re-execution are added to the dependence graph. It is in this way that the dependence graphs are dynamic. Re-executing a reader may change other data values, whose readers are then inserted into the queue. Change propagation terminates when the reader queue becomes empty.

The dynamic dependence graphs maintained by our approach depend only on the current input and not on the modifications that lead to the current input. Dynamic dependence graphs are therefore history independent [16, 20, 13]. For change propagation, however, we maintain a topological sort of the dynamic dependence graph by using the Dietz-Sleator order maintenance data structure [9], which is not history independent. For a class of algorithms we show that the topological sort as well as the dynamic dependence graphs can be maintained so that we obtain strongly history-independent dynamic algorithms [20, 13]. The tree-contraction algorithm falls in this class and therefore yields an efficient dynamic, strongly history-independent data structure for dynamic trees. The data structure is randomized. Generating a history-independent version of existing deterministic dynamic tree algorithms [25, 26, 12, 15] remains an open problem.

This paper extends our previous dynamization technique [1] by relaxing one-write per read restriction. The more important contribution, however, is trace stability, its application to the dynamic-trees problem, and history independence.

2 Related Work

Many researchers have previously considered generating dynamic solutions to problems based on their static solutions. This approach is often called *dynamization*. Bentley and Saxe [4] dynamized a class of search problems by decomposing them into parts so that an update is applied to one part (statically), and a search is applied to all parts. The approach trades off search-time for update-time. Overmars [21, 22] dynamized a class of $C(n)$-order decomposable problems by maintaining partial static solutions in a tree such that an update only propagates along a path of the tree. He used this approach, for example, to generate a $O(\log^2 n)$-time dynamic algorithm for convex hulls.

Mulmuley and independently Schwarzkopf proposed a dynamic shuffling approach that maintains a history of operations and dependences among them [19, 24]. The approach allows insertions/deletions in arbitrary positions back in "history". The effects of the update are then propagated forward in time. History, the dependences, and propagation are implemented and analyzed on a per-problem basis. Basch, Guibas, and Herschberger [3] use a similar approach to implement Kinetic Data Structures. They maintain certificates that correspond to certain predicates. As kinetic values change, these predicates can also change value, requiring the changes to be propagated through future code that depend on them. Again, predicates, dependences, and the propagation of changes are implemented on a per-problem basis. None of these approaches address how in general changes can be propagated.

Several researchers have tried techniques that can be applied automatically to a reasonably broad set of problems. Demers, Reps, and Teitelbaum [7] introduced the idea of maintaining a dependence graph of a computation, so that any change in the input can be propagated through the graph. Their approach, however, can only be applied if the dependence graph is static (oblivious to the input). Pugh and Teitelbaum [23], showed how to use memoization to dynamize static algorithms for a class of divide-and-conquer algorithms. The approach is similar to the $C(n)$-decomposable framework of Overmars [21], but works automatically. They also develop an analysis technique based on stable decompositions. The notion of stability we define in this paper is motivated by their idea. The class of problems that can be handled by memoization, however, is very different from what can be handled by our approach.

Dynamic Trees. Dynamic-tree data structures have found applications in many areas and have been studied extensively [25, 26, 12, 27, 14, 15]. Sleator and Tarjan first considered the dynamic-trees problem and presented a data structures that supports edge insertions and deletions (*aka*, the `link` and `cut` operations), re-rooting, and path queries in both amortized and worst-case $O(\log n)$ time, for a forest with n vertices [25, 26]. Cohen and Tamassia extended their technique to maintain dynamic expression trees [6]. Frederickson developed Topology Trees and applied it to various problems [11, 12]. Topology trees provide worst-case $O(\log n)$ update times for various dynamic tree operations in bounded degree trees. Holm and Lichtenberg modified Frederickson's data structure to support unbounded degree trees [15]. Henzinger and King [14] and independently Tarjan [27] developed simpler dynamic-tree data structure based on Euler-tours for a more limited interface supporting subtree queries.

History Independence. History-independent data structures preserve privacy by making it difficult or impossible to infer information about the past states of a data structure based on its current state. History-independent data structures were first studied by Micciancio, who showed that the shape of a variant of 2-3 trees is independent of the history of opera-

tions [16]. Naor and Teague [20] extended Micciancio's model to encompass the memory and defined weak and strong history independence. Strong history independence ensures privacy even under unlimited number of encroachments. Hartline *et al* [13], simplified Naor and Teague's definitions and showed that strong history independence requires canonical representation.

3 Input Changes and Trace Stability

We formalize the notions input changes, and trace stability. The stability definition is parameterized over the choice of a trace and the distance metric for traces. In Section 4, we define the particular trace and distance metric for the model we use in this paper.

Define a *trace generator* for an algorithm A, written A_T, as a version of the algorithm that outputs a trace and let $\delta(\cdot,\cdot)$ be a distance metric on traces. For example, the trace can be chosen as the function-call tree of the execution, and the trace distance can be defined as the sum of the costs of function calls that are different in two traces.

We study the similarity of traces generated by an algorithm under various classes of input changes. Formally, we define a *class of input changes* as a relation Δ, whose domain and codomain are both the input space. If $(I, I') \in \Delta$, then the modification of I to I' is a valid input change for Δ. To facilitate analysis, we partition Δ by an integer parameter n, and denote the i^{th} partition as Δ_i. Typically the parameter represents the size of the input I. For output-sensitive algorithms, Δ can be partitioned according to the output change.

Definition 3.1 (Worst-Case Stability) Let A_T be a trace generator for an algorithm A and let Δ_n be a class of input changes parameterized by n. Define

$$d(n) = \max_{(I,I')\in\Delta_n} \delta(A_T(I), A_T(I')).$$

Then A_T is S-stable for Δ_n if $d(n) \in S$.

Note that $O(f(n))$-stable, $\Omega(f(n))$-stable, and $\{f(n)\}$-stable are all valid uses of the stability notation.

For randomized algorithms, it is important that we use the same random bits on different inputs—because otherwise, traces will certainly not be stable. We therefore separate the random bits used by the algorithm from the input and denote the trace as $A_T(I, R)$, where R is the sequence of random bits. When analyzing stability we assume the same random bits are used on different inputs, and the expectation is taken over all possible choices of the random bits.

Definition 3.2 (Expected-Case Stability) Let A_T be a trace generator for a randomized algorithm A, let Δ_n be an class of input changes parameterized by n and let $\phi(\cdot)$ be a probability-density function on random bit strings $\{0,1\}^*$. Define

$$d(n) = \max_{(I,I')\in\Delta_n} \sum_{R\in\{0,1\}^*} \delta(A_T(I,R), A_T(I',R)) \cdot \phi(R).$$

Then A_T is expected S-stable for Δ_n and ϕ if $d(n) \in S$.

4 Machine Model and Traces

We describe a general-purpose machine model for automatically dynamizing static algorithms. The model is Turing-complete and any static program expressed in this model can be dynamized automatically, but the performance of the dynamized version depends on the trace stability of the program. The main results are Theorems 4.1 and 4.2, which bound the time for dynamic-updates in terms of trace stability and show the relationship to strongly history-independent data structures.

The machine model is a variant of the RAM model that places certain restriction on reading from and writing to memory. These restrictions are motivated by the need to (1) determine data dependences, (2) determine the scope of the code that needs to be re-executed when a data value changes, *i.e.*, the control dependences, and (3) re-execute code in the same state as it was previously executed. Note that when code is re-executed it can take, because of conditionals, a different path than it previously did.

Machine Model. As in the RAM model the machine consists of a random-access memory and a finite set of registers R, where $R = r_0, \ldots r_k$. The memory consists of words, which are large enough to hold a pointer. In addition to the registers, a stack pointer and a program counter are also maintained in registers that are invisible to the user.

A *program* is a set of function definitions containing a `main` function where the execution starts. Functions take no explicit parameters—all values are passed through registers. The body of a function consists of a sequence of instructions, which include the usual instructions for arithmetic (`add`, `mul`, *etc*), branches, a `write` instruction for writing to memory, and a `calln` instruction for reading from memory and making function calls. Branches are restricted to jump to a location within the function body where they appear.

The instruction `write` r_i, r_j stores the value of r_j at memory location specified by r_i. Memory can be read only during a function call, `calln`, which takes the number of locations to be read and a function. Execution of `calln` n, f saves the registers and the return address onto the stack, dereferences the first n registers (*i.e.*, $r_i \leftarrow Mem[r_i]$ for $0 \le i \le n-1$), and

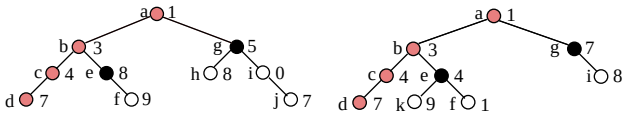

Figure 1: The traces of a program on two inputs.

jumps to the first instruction of the function. Return from the function restores the registers by popping the stack and passes the control back to return point.

The motivation for using functions for reading is to identify the code that depends on the value being read. We note that since the registers are restored when returning from a function call, the only way to return results is through memory. This ensures that all data dependences are tracked.

In this paper, we consider valid only those programs that write to each memory location at most once (*aka*, single-assignment) and read no memory location before it is written—all purely-functional programs satisfy these restrictions. This restriction is motivated by the need to re-execute code in the same context as originally executed. An alternative is to use a persistent data-structure to represent the memory [10, 8].

Traces. The *trace* of a program on some input is an ordered, rooted tree corresponding to the function-call tree of the execution of the program on that input. Each vertex represents a function call (`calln`), with the root representing the `main` function. Edges represent the caller-callee relationship—there is an edge from u to v if u calls v. Outgoing edges of a vertex are ordered with respect to the execution order. Each vertex is annotated with a *weight* equal to the running time of the function call and a list of the values read, called *read-values*, by that call. The weight of a trace T, written $w(T)$, is the weight of its root vertex. The *read-values* of a trace T, written $rval(T)$, is the read-values of its root.

Figure 1 shows two traces of some program. Each vertex is labeled with a letter and its read-value. The weight of a vertex v is the number of vertices in the subtree rooted at v—for example, the weight of b is five on the left and six on the right.

Given a trace T, we call the path from the parent of a vertex v to the root of T as the *call path* of v; the call path of the root is empty. When comparing two traces of some program, we will deem certain vertices equivalent—throughout this paper, we only compare traces belonging to the same program. Given traces T and T', we say that vertices v of T and v' of T' are *cognates* if they have the same call path relative to the root of their traces, including the read-values. For example, in Figure 1 the vertex a on the left and a on the right are cognates, and so are the b, c, d, e, g 's. Since programs in our model are deterministic, two cognates whose read-values are the same have the same number of children. We say that a vertex v is a *guard* if the read-value of its cognate v' is different than the read-values of v. In Figure 1 vertices e and g are guards. For our model, we define the distance between two traces as the sum of the weights of all guards in both traces. For example, the distance between the traces in Figure 1 is $(2+3)+(4+2)=11$. Trace distance can be defined recursively as follows.

Definition 4.1 (Trace Distance) The distance between traces T and T', $\delta(T, T')$, of a program is

$$\delta(T,T') = \begin{cases} w(T)+w(T') & \text{if } rval(T) \neq rval(T') \\ \sum_{i=1}^{k} \delta(T_i, T'_i) & \text{otherwise,} \end{cases}$$

where T_i and T'_i are the ith subtree of T and T'.

Stability and Automatic Dynamization. We present our main results for this section. The theorems rely on results from Section 6. Theorem 4.1 only applies in the worst case, whereas Theorem 4.2 applies in both expected and the worst case.

Theorem 4.1 (Stability & Dynamization) *Let A_T be the trace-generator for an algorithm A. If A_T is $O(f(n))$-stable for a class of input changes Δ, then the output of A can be updated in $O(f(n)\log f(n))$ time for any change from Δ.*

Proof. Follows directly from Lemma 6.1 by using a logarithmic-time priority queue. ■

For r-round parallel computations, we improve on this bound and also achieve strong history independence as defined by Hartline *et al.* [13]. We say that an algorithm is *r-round parallel* if it operates in r rounds where a `calln` in round i only reads memory locations written in the body of a `calln` from a round $j < i$. We say that a dynamic algorithm is *strongly history independent* if after any number of input changes and output updates the memory layout of the implementation depends only on the current input I.

Theorem 4.2 (R-Round-Parallel) *Let A_T be the trace-generator for an r-round parallel algorithm A. If A_T is (expected) $O(f(n))$-stable for some class of input changes Δ, then the output of A can be updated in (expected) $O(f(n)+r)$ time for any change from Δ. Furthermore, if each location is read by at most a constant times, then there is a strongly history independent implementation of the resulting dynamic algorithm.*

Proof. History independence follows directly from Lemma 6.2. The worst case time bound follows directly from Corollary 6.1. The expected bound follows from Corollary 6.1 by simple algebra. ■

```
tree_contract_MR (F) {
  while (#edges(F) > 0)
    for each v ∈ vertices(F)
      contract(v) }
contract(v) {
  if (v.degree = 1) // rake
    u = neighbor(v)
    if (u.degree > 1 or u > v)
      delete v
  else if (v.degree = 2) // compress
    (u1, u2) = neighbors(v)
    if (u1.degree > 1 and u2.degree > 1 and
        flips(u1, v, u2)=(T, H, T))
      connect u1 and u2
      delete v }
```

```
tree_contract {
  for i = 1 to k log n do
    for j = 1 to n do
      // r0 ← &A[i, j].live; r1 ← &A[i, j].degree
      // r2 ← &A[i, j].ngh[0] ... r_{t+2} ← &A[i, j].ngh[t]
      calln (t + 3), contract }
contract {
  if (not live)
    write &A[i + 1, j].live,false
  else if (degree = 1)
    rake ...
  else if (degree = 2)
    compress ...
  else
    find degree & singleton status, copy ...  }
```

Figure 2: Randomized Tree Contraction (left) and Tree Contraction in our model (right)

5 Tree Contraction and Dynamic Trees

We dynamize the parallel tree-contraction algorithm of Miller and Reif [17] to obtain a data structure for dynamic-trees [25, 26, 12, 15], which we call RC (Rake-and-Compress) Trees. We first describe parallel tree contraction, and then show that it is expected $O(\log n)$-stable with respect to edge insertions and deletions. Theorem 4.2 implies an expected $O(\log n)$ time, history-independent data structure for dynamic trees.

5.1 Parallel Tree Contraction Given a t-ary forest, the parallel tree-contraction algorithm operates in rounds. In each round, each tree is contracted by applying the rake and compress operations. The rake operations delete all leaves of the tree (if the tree is a single edge, then only one leaf is deleted). The compress operations delete an independent set of degree-two vertices. All rake and compress operations within a round are local and are applied "in parallel" (*i.e.*, all decisions are based on the state when the round starts). The algorithm terminates when no edges remain.

Various versions of tree contraction have been proposed depending on how they select the independent set of degree-two vertices to compress. There are two basic approaches, one deterministic and the other randomized. We use a randomized version—each vertex flips a coin in each round and a degree-two vertex is compressed if it flips a head, its two neighbors both flip tails, and neither neighbor is a leaf. This compress rule is a slight generalization of the original rule by Miller and Reif [17], which only applies to rooted trees. Figure 2 shows pseudo-code for randomized tree contraction. It is not hard to show that the algorithm takes logarithmic number of rounds in the expected case.

Using tree contraction the programmer can perform various computations on trees by associating data with edges and vertices and defining how data is accumulated during rake and compress [17, 18]. In a related paper [2], we describe how various properties of trees can be maintained dynamically using RC-Trees. We consider a broad set of applications including path queries, subtree queries, centers/medians/diameters of trees, shortest distance to a set of marked nodes, and least common ancestor queries [2]. We also describe how to separate handling of application-specific data and queries from structural changes (edge insertions and deletions). We therefore consider only structural changes in this paper.

Implementation. Figure 2 shows the pseudo code for an implementation of tree contraction in our model. Since the model requires that a location be written at most once, we copy the forest in each round. For a t-ary forest with n vertices, we keep the vertices in an array A with $n \times (k \log n + 1)$ cells (k is a constant to be specified in Theorem 5.2). The input forest is stored in the first row, and tree contraction is performed for $k \log n$ rounds—the i^{th} round reads the i^{th} row and writes to $(i+1)^{st}$ row. Each cell of the array represents an instance of a vertex and consists of a live flag, a degree field, a singleton-status flag, and neighbor pointers. The *singleton status* of a vertex is a binary value indicating if that vertex is a leaf or not.

Each round calls the function `contract` on all vertices. The call to contract reads the liveness flag, the de-

gree, and the neighbor pointers for that vertex into registers. Rake and compress operations are implemented as with the original algorithm except that we make another `calln` in order to read the singleton status of each neighbor. If a vertex is live and is not deleted in this round, we compute its degree based on the singleton status of its neighbors, compute its singleton-status, and copy it to the next round.

The trace for tree-contraction is a tree with height three. The root has $nk \log n$ children, each of which corresponds to a call to `contract` and is either a leaf or has one child. Each depth-three vertex corresponds to a call for reading the singleton status of neighbors. Each vertex except for the root has a constant weight.

To generate random coin flips, we use a family H of 3-wise independent hash functions mapping $\{1 \dots n\}$ to $\{0,1\}$. We randomly select $k \log n$ members of H, one per round. Since there are families H with $|H|$ polynomial in n [28], we need $O(\log^2 n)$ random bits. As discussed in Section 3 we analyze trace stability assuming fixed selection of random bits and take expectations over all selections of bits. For fixed random bits, vertex i will generate the same coin flip on round j for any input. This ensures that the trace remains stable in different executions.

After $k \log n$ rounds, it is possible, though with small probability, that not all edges are deleted. When that happens, we can use any linear-time tree evaluation algorithm to compute the result.

This implementation is $k \log n$-round parallel as defined in Section 4, and every location is read at most $(t+1)$ times, therefore Theorem 4.2 applies.

The implementation as described requires $O(n \log n)$ memory. Since with high probability only $O(n)$ locations will contain live vertices [17] we can use a hash table of size $O(n)$ to implement the memory in our model. Furthermore we don't need to store the dependences between non-live locations since they all go from $A[i,j]$ to $A[i+1,j]$. Therefore the algorithm can be implemented in $O(n)$ space—it is not clear, however, if this implementation would be strongly history independent.

5.2 Trace Stability of Parallel Tree Contraction

We analyze the trace stability of tree contraction under a single edge deletion or insertion. Data changes and queries can be handled separately with support trees [2].

Definitions. Consider a forest $F = (V, E)$ with n vertices and execute tree-contraction on F. Order vertices in the order that they are processed in each round $V = (v_1, \dots, v_n)$. In what follows, the term *"at round i"* means, *"at the beginning of round i"*. We denote the contracted forest at round i as $F^i = (V^i, E^i)$. A vertex v is *live* at round i, if $v \in V^i$—it has not been deleted (compressed or raked) in some previous round. We define the *configuration* of v at round $i \geq 1$

$$\kappa_F^i(v) = \begin{cases} \{(u, \sigma(u)) \mid (v,u) \in E^i\} & v \in V^i \\ \mathsf{deleted} & \text{otherwise} \end{cases}$$

Here $\sigma(u)$ is the singleton status of vertex u (whether u has degree one or not). Configuration of a vertex represents the values read by the call to `contract`.

We define a *contraction* $\mathcal{X}_F$ of F to be $\mathcal{X}_F = \kappa_F^1(v_1) \dots \kappa_F^1(v_n)\, \kappa_F^2(v_1) \dots \kappa_F^{k \log n}(v_n)$. A contraction serves as an abstraction of the trace.

Consider some forest $G = (V, E \setminus \{(u,v)\})$ obtained from F by deleting the edge (u,v) and let $\mathcal{X}_G$ be its contraction. Let T_F and T_G be traces for the contractions $\mathcal{X}_F$ and $\mathcal{X}_G$. The distance between T_F and T_G is within a factor of two of the difference between $\mathcal{X}_F$ and $\mathcal{X}_G$, that is $\delta(T_F, T_G) = O\left(\sum_{i=1}^{k \log n} \sum_{j=1}^{n} \mathsf{neq}(\kappa_F^i(v_j), \kappa_G^i(v_j))\right)$, where $\mathsf{neq}(\cdot,\cdot)$ is one if the configurations are not equal and zero otherwise. Thus to bound the trace distance it suffices to count the places where two configurations do not match in $\mathcal{X}_F$ and $\mathcal{X}_G$.

We say that a vertex v *becomes affected in round i*, if i is the earliest round for which $\kappa_F^{i+1}(v) \neq \kappa_G^{i+1}(v)$. Once a vertex becomes affected, it remains *affected* in any subsequent round. Only input changes will make a vertex affected at round one. We say that a vertex v is a *frontier* at round i, if v is affected and is adjacent to a unaffected vertex at round i. We denote the set of affected vertices at round i as A^i—note that A^i includes all affected vertices live or deleted. For any $A \subseteq A^i$, we denote the forests induced by A on F^i and G^i as F_A^i and G_A^i respectively. Since all deleted vertices have the same configuration $\delta(T_F, T_G) = O\left(\sum_{i=1}^{k \log n} |F_{A^i}^i| + |G_{A^i}^i|\right)$.

An *affected component* at round i, $\mathcal{AC}^i$ is defined as a maximal set satisfying (1) $\mathcal{AC}^i \subseteq A^i$, and (2) $F_{\mathcal{AC}^i}^i$ and $G_{\mathcal{AC}^i}^i$ are trees.

Analysis. To prove that tree contraction is expected $O(\log n)$ stable we will show that the expected size of $F_{A^i}^i$ and $G_{A^i}^i$ is constant at any round i. The first lemma establishes two useful properties that we use throughout the analysis.

Lemma 5.1 *(1) A frontier is live and is adjacent to the same set of unaffected vertices at any round i in both $\mathcal{X}_F$ and $\mathcal{X}_G$. (2) If a vertex becomes affected in any round i, then it is either adjacent to a frontier or it has neighbor that is adjacent to a frontier at that round.*

Proof. The first property follows from the facts that (A) a frontier is adjacent to a unaffected vertex at

that round, and (B) unaffected vertices have the same configuration in both $\mathcal{X}_F$ and $\mathcal{X}_G$. For the second property, consider some vertex v that is unaffected at round i. If v's neighbors are all unaffected, then v's neighbors at round $i+1$ will be the same in both $\mathcal{X}_F$ and $\mathcal{X}_G$. Thus, if v becomes affected in some round, then either it is adjacent to an affected vertex u, or v has neighbor u whose singleton status differs in $\mathcal{X}_F$ and $\mathcal{X}_G$ in round $(i+1)$, which can happen only if u is adjacent to an affected vertex. ■

We now partition A^i into two affected components $\mathcal{AC}^i_u$ and $\mathcal{AC}^i_v$, $A^i = \mathcal{AC}^i_u \cup \mathcal{AC}^i_v$, such that $G^i_{A^i} = G^i_{\mathcal{AC}^i_u} \cup G^i_{\mathcal{AC}^i_v}$ and $F^i_{A^i} = F^i_{\mathcal{AC}^i_u \cup \mathcal{AC}^i_v}$. Affected components correspond to the end-points of the deleted edge (u, v). Note $G^i_{\mathcal{AC}^i_u}$ and $G^i_{\mathcal{AC}^i_v}$ are disconnected.

Lemma 5.2 *At round one, each of $\mathcal{AC}^1_u$ and $\mathcal{AC}^1_v$ contain at most two vertices and at most one frontier.*

Proof. Deletion of (u, v) makes u and v affected. If u does not become a leaf, then its neighbors remain unaffected. If u becomes a leaf, then its neighbor u' (if it exists) becomes affected. Since u and v are not connected in G, the set $\{u\}$ or if u' is also affected $\{u, u'\}$ is an affected component, this set will be $\mathcal{AC}^1_u$. Either u or if u' exists, then u' may be a frontier. The set $\mathcal{AC}^1_v$ is built by a similar argument. ■

Lemma 5.3 *Assume that $\mathcal{AC}^i_u$ and $\mathcal{AC}^i_v$ are the only affected components in round i. There are exactly two affected components in round $i+1$, $\mathcal{AC}^{i+1}_u$ and $\mathcal{AC}^{i+1}_v$.*

Proof. By Lemma 5.1, we consider vertices that become affected due to each frontier. Let U and V be the set of vertices that become affected due to a frontier in $\mathcal{AC}^i_u$ and $\mathcal{AC}^i_v$ respectively and define $\mathcal{AC}^{i+1}_u = \mathcal{AC}^i_u \cup U$ and $\mathcal{AC}^{i+1}_v = \mathcal{AC}^i_v \cup V$. This accounts for all affected vertices $A^{i+1} = A^i \cup U \cup V$. By Lemma 5.1 the vertices of U and V are connected to some frontier by a path. Since tree-contraction preserves connectivity, $F^{i+1}_{\mathcal{AC}^{i+1}_u}, F^{i+1}_{\mathcal{AC}^{i+1}_v}$ and $G^{i+1}_{\mathcal{AC}^{i+1}_u}, G^{i+1}_{\mathcal{AC}^{i+1}_v}$ are trees, and $G^{i+1}_{\mathcal{AC}^{i+1}_u}, G^{i+1}_{\mathcal{AC}^{i+1}_v}$ remain disconnected. ■

By induction on i, using Lemmas 5.2 and 5.3, the number of affected components is exactly two. We now show that each affected component has at most two frontiers. The argument applies to both components, thus we consider some affected component $\mathcal{AC}$.

Lemma 5.4 *Suppose there is exactly one frontier in $\mathcal{AC}^i$. At most two vertices become affected in round i due to contraction of vertices in $\mathcal{AC}^i$ and there are at most two frontiers in $\mathcal{AC}^{i+1}$.*

Proof. Let x be the sole frontier at round i. Since x is adjacent to a unaffected vertex, it has degree at least one. Furthermore x cannot have degree one, because otherwise its leaf status would be different in two contraction making its neighbor affected. Thus x has degree two or greater.

Suppose that x is compressed in round i in $\mathcal{X}_F$ or $\mathcal{X}_G$. Then x has at most two unaffected neighbors y and z, which may become affected. Since x is compressed, y will have degree at least one after the compress, and thus no (unaffected) neighbor of y will become affected. Same argument holds for z. Now suppose that x is not deleted in either $\mathcal{X}_F$ or $\mathcal{X}_G$. If x does not become a leaf, then no vertices become affected. If x becomes a leaf, then it has at most one unaffected neighbor, which may become affected. Thus, at most two vertices become affected or frontier. ■

Lemma 5.5 *Suppose there are exactly two frontiers in $\mathcal{AC}^i$. At most two vertices become affected in round i due to the contraction of vertices in $\mathcal{AC}^i$ and there are at most two frontiers in $\mathcal{AC}^{i+1}$.*

Proof. Let x and y be two frontiers. Since x and y are in the same affected component, they are connected by a path of affected vertices and each is also adjacent to a unaffected vertex. Thus each has degree at least two in both contractions at round i. Assume that x is compressed in either contraction. Then x has at most one unaffected neighbor w, which may become affected. Since w has degree at least one after the compress, no (unaffected) neighbor of w will become affected. If x is not deleted in either contraction, then no vertices will become affected, because x cannot become a leaf—a path to y will remain, because y cannot be raked. Therefore, at most one vertex will become affected and will possibly become a frontier. The same argument applies to y. ■

Lemma 5.6 *The total number of affected vertices in F^i and G^i is $O(1)$ in the expected case.*

Proof. Consider applying tree contraction on F^i and note that $\mathcal{F}^i_{A^i}$ will contract by a constant factor when we disregard the frontier vertices, by an argument similar to that of Miller and Reif [17]—based on independence of randomness between different rounds. The number of affected components is exactly two and each component has at most two frontiers and cause two vertices become affected (by Lemmas 5.2, 5.4, and 5.5). Thus, there are at most four frontiers and four new vertices may become affected in round i. Thus, we have $E[|F^{i+1}_{A^{i+1}}|] \leq (1-c)|\mathcal{F}^i_{A^i}| + 8$ and $E[|F^{i+1}_{A^{i+1}}|] \leq (1-c)E[|\mathcal{F}^i_{A^i}|] + 8$.

Since the number of affected vertices in the first round is at most 4, $E[|F^i_A|] = O(1)$, for any i. A similar argument holds for G^i. ■

Theorem 5.1 *Tree contraction is expected $O(\log n)$ stable for a single edge insertion or deletion.*

Proof. By Lemma 5.6, the expected number of live affected vertices is constant in every round. Since tree-contraction takes $k \log n$ rounds, the result follows by the linearity of sums of expectations. ■

Theorem 5.2 *Tree contraction algorithm yields a strongly history-independent, expected $O(\log n)$-time data structure for the dynamic trees problem for insertions and deletions of edges.*

Proof. The tree contraction algorithm is $k \log n$-round-parallel as defined in Section 4. Since each location is read at most a constant number of times, the algorithm is strongly history independent and takes expected $O(\log n)$ time by Theorems 4.2 and 5.1.

Since a constant fraction of all vertices are deleted in each round with a nonzero constant probability, the probability that the contraction is not complete (the forest contains an edge) after $k \log n$ rounds is at most $1/n$ for some constant k [17]. Therefore running a linear time tree evaluation algorithm to finish off the contraction takes $O(1)$ expected time. ■

6 Automatic Dynamization

This section presents a constructive proof of Theorems 4.1 and 4.2 by using dynamic dependence graphs and change propagation.

Dynamic Dependence Graphs. A dynamic dependence graph is built by the execution of a program in order to represent the data and control dependences in that execution. It contains two kinds of edges, trace edges, which correspond to control dependences, and data-dependence edges. Each vertex of a dynamic dependence graph is also tagged with a time stamp that corresponds to the execution time of that vertex and also other information to facilitate change propagation.

Formally, a *dynamic dependence graph* $DDG = ((V, L), (E, D))$ consists of vertices partitioned into trace vertices V and locations L, and a set of edges partitioned into trace edges $E \subseteq V \times V$ and *dependences* $D \subseteq L \times V$. The graph (V, E), which we call a *trace*, is a function call tree and is structurally equivalent to the trace defined in Section 4—unlike the original definition, however, we do not require that the tree be ordered. The dependences D corresponds to data dependences between locations and function calls. The function $\mathcal{V}$ maps each vertex in V to a quadruple consisting of (1) a time-stamp, (2) the function being called, (3) the values of registers immediately before the function call, and (4) the number of reads. The function M, called the *memory* maps each location to a value. A dynamic dependence graph along with memory M and labels $\mathcal{V}$ constitutes all the state needed by change propagation.

Time stamps maintain a topological ordering of the trace (V, E). During change propagation, functions affected by the changes are re-executed in this topological order. Since both control and data dependences go from earlier to later function calls, propagating changes in topological order is critical for correctness. Since dynamic dependence graphs change during change propagation, order of time-stamps must be maintained dynamically. For this, we use the Dietz-Sleator order-maintenance algorithm [9], which supports constant time insertion, deletion, and comparison operations.

Building a Dynamic Dependence Graph. A dynamic dependence graph is built by execution and kept up to date by change propagation. In addition to the set of location L and the memory M, we keep track of the set of *changed locations* L_C; L_C will be used by the change propagation algorithm. We assume that a set of memory locations are given as inputs I. Initially $L = I$ and M maps each input location to a value. Each `write` l, v instruction sets $M(l)$ to v. If $l \notin L$, then L is extended with l ($L = L \cup \{l\}$); otherwise, l is added to the set of changed locations ($L_C = L_C \cup \{l\}$). Since programs in our model are single-assignment, a location will never be added to L_C in the initial execution.

The executed `calln` instructions build the trace and the dependence edges. Initially V, E, D are all set to the empty set and the current time-stamp is initialized to the "beginning of time". Execution of a `calln` n, f instruction (1) extends V with a new vertex v ($V = V \cup \{v\}$), (2) inserts a trace edge from the caller u to v ($E = E \cup \{(u, v)\}$), (3) inserts a dependence edge from locations being read L_R to v ($D = D \cup \{(l, v) \mid l \in L_R\}$) (4) advances the time by creating a new time-stamp t after the current time and setting the current time to t, (5) extends $\mathcal{V}$ by mapping v to the current time, the value of the registers, the function being called, and the number of locations read, (6) dereferences the registers and makes the call.

Change Propagation. The pseudo-code for change propagation is shown in Figure 3. The algorithm takes as input a set of input changes C, the memory M, and a dynamic dependence graph DDG. The *input changes* C maps a set of locations to new values. The algorithm updates the values of changed locations in M (line 2), and builds a priority queue of affected vertices reading the changed locations (line 3). A vertex is called *affected* if the value of a location that it reads has changed since the last execution.

Each iteration of the while loop re-executes a guard vertex v. First the subtree T^- for the previous execu-

```
propagate (M, C, DDG as ((V, L), (E, D)))
1  T = (V, E)
2  M = (M \ {(l, M(l)) | l ∈ C}) ∪ C
3  Q = {v | (l, v') ∈ C, (l, v) ∈ D}
4  while Q ≠ ∅
5    v = deleteMin(Q)
6    T⁻ = subtree rooted at v
7    (n, fun, time, R) = V(v)
8    ((V⁺, E⁺), D⁺, L_C) = calln n, fun at time
                  with regs R and Memory M
9    T⁺ = (V⁺, E⁺)
10   T' = replace T⁻ with T⁺ in T
11   D⁻ = {(l, u) | u ∈ vertices(T⁻) ∧ (l, u) ∈ D}
12   D = D ∪ D⁺ \ D⁻
13   A = {u | l ∈ L_C ∧ (l, u) ∈ D}
14   Q = Q ∪ A \ vertices(T⁻)
15   ∀u ∈ vertices(T⁻), delete time-stamp(u)
```

Figure 3: The change-propagation algorithm.

tion of the call is determined. Then the function of v is re-executed starting at the same time and with the same registers as it was previously executed (line 8). Let $T^+ = (V^+, E^+)$ denote the trace and D^+ denote the data-dependences for re-execution, and L_C be the set of locations changed during re-execution. The trace is updated by replacing the subtree T^- rooted at v with T^+ (line 10) and dependences are updated by adding D^+ to and deleting D^- from D (line 12)—D^- is the dependences pertaining to T^-. The queue is updated by removing the vertices of T^- and inserting the newly affected vertices A (line 14). Finally the time-stamps of the vertices of T^- are deleted.

Re-executing function calls in topologically sorted order according to their time stamps serves two purposes. First, it ensures that function calls that are not relevant to the computation are not taken. For example, in Figure 1, change propagation that starts with the trace on the left and yields the trace on the right should not re-execute the call h; indeed h is removed from the queue when g is re-executed (line 14). Second, it ensures that a value read by a `calln` is up-to-date, because such a value may depend on a previous call.

Although the sequential execution order gives a topological sort of a dynamic dependence graph, other forms of topological ordering can be more efficient. For example, in an r-round-parallel computation, all edges are from earlier to later rounds. Thus we can use round numbers as time stamps and use array-based, constant-time priority queues.

Implementation and Performance. We implement dynamic dependence graphs by using a standard tree representation and by maintaining for each memory location a list of pointers to vertices reading that location and back. For change propagation, we obtain different bounds depending on the priority queue we employ. Our main lemma is therefore parameterized over the time for priority queue operations.

Lemma 6.1 *Consider the traces $T(I)$ and $T(I')$ of a program on inputs I and I' and let $d = \delta(T(I), T(I'))$. Change propagation takes time $O(d + d \cdot p(d))$ with a priority queue that supports insertion, deletion, and delete-minimum operations in $O(p(\cdot))$ time.*

Proof. Consider an iteration of the change propagation loop and let v be the vertex removed from the queue. It is a property of the algorithm v is a guard with cognate v'. Let w and w' denote weights of v and v' respectively. Determining T^- takes constant time (line 6). Re-executing the function call takes time w' (line 8). Updating the trace takes constant time (*line* 10). Since each function call performs a constant number of reads, computing D^- takes $O(w)$ time. Since D^+ has $O(w')$ dependences, updating the dependences takes in $O(w + w')$ time (line 12). Deleting the time-stamps for vertices of T^- takes time $O(w)$. Thus, an iteration of the loop takes time $O(w+w')$ time plus time for finding the affected vertices (line 13) and priority queue operations (lines 5 and 14).

Since re-execution of a vertex removes its descendants from the queue (line 14), only guards are re-executed. Since the trace-distance is the total weights of the guards, the total time is $O(d)$ plus time for finding the affected vertices and priority queue operations.

Let m be the total number of affected vertices. Finding all affected vertices (line 13) takes time $O(m)$. Since each vertex is added to and removed from the priority queue once, the time for all priority queue operations is $O(m \cdot p(m))$. Since the weight of a vertex is no less than the number of its descendants, $m \leq d$. Therefore the total time is $O(d + dp(d))$. ■

For r-round parallel computations we use round numbers as time stamps and implement priority queues with r buckets supporting all operations in $O(1)$ time.

Corollary 6.1 *Let $T(I)$ and $T(I')$ be the traces of an r-round parallel program on inputs I and I' and $d = \delta(T(I), T(I'))$. Change propagation takes $O(d + r)$ time.*

Lemma 6.2 *Consider an r-round-parallel algorithm and suppose the algorithm reads each memory location some constant number of times on all inputs. The algorithm can then be dynamized to yield a strongly history-independent dynamic algorithm.*

Proof. Hartline *et al* [13] show that a data structure is strongly history independent if and only if it has a canonical memory representation—every input has a unique layout in memory. Thus, to show that an automatically dynamized algorithm is strongly history independent, we show that dynamic dependence graphs have a canonical representation.

The structure of the dynamic dependence graph is clearly canonical, but the actual memory layout might not be. Since the number of reads of each location is constant, we can store the information associated with any vertex v (`calln`) in memory associated with the first location that it reads (we inline calls with no reads so that they don't have a vertices in the trace). To maintain the children of v, we will have a list that runs through the children themselves. Since a location is read constant times, the dependence edges and the vertices can be stored in sorted order based on, for example, the bit representation. Finally, the structure of the order maintenance algorithm we rely on [9] is not canonical. For r-round parallel computations, however, we can use the round numbers as time stamps. ■

Acknowledgments

We thank Bob Tarjan and Renato Werneck for many discussions and feedback.

References

[1] 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.

[2] U. A. Acar, G. E. Blelloch, and J. L. Vittes. Separating structure from data in dynamic trees. Technical report, Department of Computer Science, Carnegie Mellon University, 2003.

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

[4] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[5] M. Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 26–35. ACM Press, 2002.

[6] R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–61, 1991.

[7] 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.

[8] P. F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer-Verlag, August 1989.

[9] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings. 19th ACM Symposium. Theory of Computing*, pages 365–372, 1987.

[10] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, Feb. 1989.

[11] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14:781–798, 1985.

[12] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *J. Algorithms*, 24(1):37–65, 1997.

[13] J. D. Hartline, E. S. Hong, A. E. Modht, W. R. Pentney, and E. C. Rocke. Characterizing history independent data structures. In *Proceedings of Thirteenth International Symposium on Algorithms and Computation (ISAAC)*, volume 2518 of *Lecture Notes in Computer Science*. Springer, 2002.

[14] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[15] J. Holm and K. de Lichtenberg. Top-trees and dynamic graph algorithms. Technical Report DIKU-TR-98/17, Department of Computer Science, University of Copenhagen, Aug. 1998.

[16] D. Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 456–464, 1997.

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

[18] G. L. Miller and J. H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[19] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling (extended abstract). In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 180–196, 1991.

[20] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 492–501. ACM Press, 2001.

[21] M. H. Overmars. Dynamization of order decomposable set problems. *Journal of Algorithms*, 2:245–260, 1981.

[22] M. H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[23] 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.

[24] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 197–206, 1991.

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

[26] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[27] R. E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:167–177, 1997.

[28] M. N. Wegman and L. Carter. New classes and

applications of hash functions. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 175–182, 1979.