Memory Management for Self-Adjusting Computation

Matthew A. Hammer Umut A. Acar *
Toyota Technological Institute at Chicago
{hammer,umut}@tti-c.org

Abstract

The cost of reclaiming space with traversal-based garbage collection is inversely proportional to the amount of free memory, i.e., $O(1/(1-f))$, where f is the fraction of memory that is live. Consequently, the cost of garbage collection can be very high when the size of the live data remains large relative to the available free space. Intuitively, this is because allocating a small amount of memory space will require the garbage collector to traverse a significant fraction of the memory only to discover little garbage. This is unfortunate because in some application domains the size of the memory-resident data can be generally high. This can cause high GC overheads, especially when generational assumptions do not hold. One such application domain is self-adjusting computation, where computations use memory-resident execution traces in order to respond to changes to their state (e.g., inputs) efficiently.

This paper proposes memory-management techniques for self-adjusting computation that remain efficient even when the size of the live data is large. More precisely, the proposed techniques guarantee $O(1)$ amortized cost for each reclaimed memory object. We propose a set of primitives for self-adjusting computation that support the proposed memory management techniques. The primitives provide an operation for allocating memory; we reclaim unused memory automatically.

We implement a library for supporting the primitives in the C language and perform an experimental evaluation. Our experiments show that the approach can be implemented with reasonably small constant-factor overheads and that the programs written using the library behave optimally. Compared to previous implementations, we measure up to an order of magnitude improvement in performance and up to a 75% reduction in space usage.

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

General Terms Languages, Performance, Algorithms.

Keywords Self-adjusting computation, memory management, garbage collection, memoization, dynamic dependence graphs, dynamic algorithms, computational geometry, performance.

* Acar is partially supported by a gift from Intel.

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.

1. Introduction

In many application domains it is often necessary for computations to respond to small changes to their data by incrementally updating their output instead of recomputing from scratch; this is broadly referred as *incremental computation*. Automatic incrementalization concerns the transformation of programs into incremental programs that can respond to changes to their input data efficiently (e.g., [15, 31], for a survey see [32]). Recent work on self-adjusting computation (e.g., [4]) broadened the applicability and efficiency of automatic incrementalization. The approach has been applied to a number of applications including invariant checking and program verification [33], motion simulation [6, 7], machine learning [8], and scientific computing [22]. For some applications, it has been shown that the approach can achieve theoretically optimal updates and even help make progress on open problems (e.g., [22, 8, 6]).

In self-adjusting computation, the programmer uses several primitives for writing programs that can automatically respond to changes to their data. The primitives enable the programmer to mark the *changeable data*, i.e., the computation data that can change over time, by storing them in so-called *modifiable references* and by reading from and writing into these modifiable references. A self-adjusting program has two modes of operation: from-scratch execution and change propagation. Like an ordinary program, a self-adjusting program can be executed with a given input to obtain an output for that input. After a from-scratch run is completed, any of the changeable data stored in modifiable references can be changed (by writing new values into the modifiable references) and the output can be updated by running a built-in *update* operation. Typically, a self-adjusting computation starts with a from-scratch execution and is followed by a number of change-and-update steps.

To achieve efficient updates, a self-adjusting computation tracks all operations on modifiables by recording them in an execution *trace*. When the data is changed and the update operation is executed, a *change-propagation algorithm* uses the trace to identify the modifiable references that are affected by the changes and updates their values by re-executing the function invocations that read them. When a function invocation is re-executed, it can change the contents of other modifiable references. The process continues until all affected references are updated and all the function invocations that read them are re-executed. When re-executing a function invocation, nested function invocations that are not affected by the changes are identified via memoization. Change propagation essentially updates the computation as if the program were executed from scratch on the new input while simultaneously reusing as much of the old computation as possible. To achieve efficient update times, traces are represented by using a from of dynamic dependence graphs [5].

Detailed practical evaluations of self-adjusting computation show that garbage collection can be a significant cost [4]. As an example, Figure 1 shows timings for from-scratch runs and change

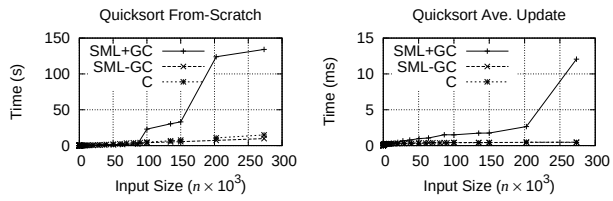

Figure 1. Comparison with `quicksort`.

propagation of a self-adjusting program. The program, a standard quicksort implementation, is taken directly from the currently available implementation of self-adjusting computation in Standard ML (SML). The program is compiled with the MLton compiler [37] and run with a fixed heap of two gigabytes. [1] The MLton compiler employs a mix of copying, generational, and mark-and-compact strategies based on the workload to deliver efficient garbage collection. The graph on the left shows the running time of a from-scratch execution of the program on inputs with up to 275,000 elements. The graph on the right shows the average time for change propagation under an insertion/deletion of an element (average taken over all possible positions in the input). The lines labeled "SML+GC" show the time including the GC time and the lines labeled "SML-GC" shows the time excluding the GC time. In both graphs, GC time increases significantly with the input size. As we describe later, the line labeled "C" shows timings with the approach described in this paper.

Self-adjusting programs are challenging for GC because

- the total amount of live data at any time can be high,
- many allocated objects have long life spans, and
- old and newly allocated data interact in complex ways.

The total amount of live data is large because a self-adjusting computation stores the trace, which records the dependences between computation data, in memory. As a result, even for moderately-sized inputs the total amount of live data can be quite high. For example, with the SML quicksort example, when the input has 100,000 elements, the amount of live data can be as high as 650 megabytes. Trace elements tend to be long-lived because change-propagation is generally successful in reusing them. Finally, change propagation makes parts of the old trace point to the newly allocated parts and vice versa, establishing cyclic dependences between old and new data.

These characteristics make self-adjusting programs pessimal for memory-traversing collectors. To see this, recall that for traversal-based GC, the cost for each reclaimed memory cell is $(1/(1-f))-1$, when f fraction of memory is live (e.g., [23]). This function grows very quickly because $1/(1-f) = 1 + f + f^2 + f^3 \ldots$. As f approaches one (memory fills) the quantity approaches infinity. Generational collectors deal with this problem by separating the data into generations to avoid traversing all of memory. Self-adjusting computation, however, does not observe the generational hypothesis (e.g., [23] chapter 7) making it unlikely that generational techniques will help. We confirmed this correlation between f and the GC costs by collecting some GC statistics. Figure 2 shows the GC cost for each allocated byte during the change-propagation experiments with quicksort (Figure 1). We measure the GC cost as the total number of bytes traversed by GC during the experiment divided by the total number of bytes allocated during the experiment. As can be seen, as the input size increases (requiring more live memory to hold the trace), the cost increases dramatically, essentially following the function $1/(1-f)$. We note that reference counting may perform better but it has larger overheads because it typically requires maintaining counters and frequently updating them. Furthermore, reference counting requires additional facilities for dealing with cycles in memory.

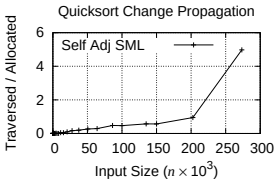

Figure 2. GC cost.

Our contributions

We describe and evaluate memory-management techniques for self-adjusting programs. We take advantage of the specifics of self-adjusting programs to manage memory such that garbage collection requires $O(1)$ amortized time per reclaimed memory cell, regardless of how much live data there is. As described above, traversing collectors require $O(\frac{1}{1-f})$ time when f fraction of the memory is live. We collect all garbage accurately (including cycles) and do not require reference counting. In our approach the programmer allocates memory explicitly using our allocator and we perform automatic garbage collection. We require that the programmer use our allocator for allocating the modifiable references and their contents (all changeable data). For other computation data, the programmer can use other allocators, e.g., C's own allocator (`malloc`, and `free`). In our experiments, we use our allocator for all allocations.

A modifiable reference and the data accessible from their contents may become garbage only when the trace of the computation changes; this is because all such data is accessible through the trace. As a result, data allocated through our allocator becomes garbage only during change propagation. To accurately and quickly collect garbage, we integrate our garbage-collection algorithm with tracing and change propagation. To do this, we enrich the traces so that programs record the *owner* of each allocated memory block, which is roughly defined as the function invocation that allocates it. As it executes, the change propagation algorithm identifies some of the function invocations and invalidates them. When function calls become invalidated the memory blocks owned by them are guaranteed to become garbage at the end of change propagation. This is due to a consistency or contextual-equivalence theorem between change propagation and from-scratch runs [3]: the theorem proves that change-propagation yields the same computation as a from-scratch run. Therefore, a function invocation that is invalidated during change propagation does not take place in the from-scratch execution and any memory allocated by it can be reclaimed when change propagation completes.

We describe an interface for writing self-adjusting programs (Section 2) and algorithms for implementing the interface (Section 3). The interface is similar to previous proposals except that (1) we do not require the user provide explicit memoization annotations—we memoize functions automatically, (2) we do not require the user to provide equality functions for modifiables—we use a physical equality instead, and (3) we supply primitives for reusing locations based on programmer-supplied indices of keys. The first two extensions simplify the interface and reduce the potential for errors. The third extension avoids the need for higher-order "lift" operators required in the previous work [4].

To evaluate the practical effectiveness of our approach, we implement a library for self-adjusting computation in the C language and a number of benchmarks which are described in Section 4. These include benchmarks used in the previous work [4] plus additional computational-geometry and tree applications. We compare our library to the most current implementation of self-adjusting computation, which is in Standard ML. Our comparisons (Section 5) show that our implementation is consistently faster than the SML implementation. When the input sizes increase, the perfor-

[1] More details on the experimental setup are given in Section 5.1.

mance gap between our implementation and the SML implementation increases, often dramatically. For example, in Figure 1, line "C" shows the timings for the same application with our C library. As can be seen, the times for the from-scratch run and change propagation are each comparable to the SML implementation when excluding the time for GC. When GC times are included, our implementation is a up to a factor of 10 faster. Our experiments also show that our implementation uses between 30% and 75% less space that the SML implementation

In addition to comparing to SML, we compare our self-adjusting benchmarks to *static* C versions. We obtain a static version of an application by replacing the modifiable references with ordinary references. This turns off all tracing and change propagation mechanisms. Our experiments show that from-scratch runs of self-adjusting programs are about two to three times slower than these static versions on average. Although self-adjusting programs can be slower when executed from scratch, they respond to changes extremely quickly. In our experiments, we measure speedups of more than four orders of magnitude over recomputing from scratch when responding to a small change such as an insertion or deletion. The speedups are large because there is often an asymptotically linear gap between the time for change propagation and executing from scratch.

2. Programming Model

In this section we describe the interface of our library for writing self-adjusting programs in C.

2.1 The interface

We provide the following set of primitives for writing self-adjusting programs in C:

- `new`(*size*, *initfunc* [, *keys*]): Allocates *size* bytes, initializes these bytes using the initialization function *initfunc* and returns its location.
- `modref`([*keys*]): Allocates a modifiable and returns its location.
- `write`(l,v): Writes value v into the modifiable at l.
- `read`(l): Reads the value of the modifiable at l. This operation may only appear as an argument to `call`.
- `call`(f, *args*): Calls function f with arguments *args*. Each argument must be a word-sized value (e.g., an integer, a pointer, *etc.*) or the read of a modifiable at l, written as `read`(l).

For memory allocation, we provide the `new` and `modref` primitives. The `new` primitive allocates memory blocks of the given size. Blocks allocated with `new` are initialized using the provided initialization function. If the initializer requires additional values from the allocator (e.g., to fill the fields of a structure being initialized) then these values must be passed as keys. The primitive `modref` allocates modifiables. When allocating modifiable, the user can optionally supply keys, each of which is a word-sized value (e.g., an integer, a pointer, etc.).

After being allocated with `new`, blocks can be dereferenced using conventional C code. After being allocated with `modref`, modifiables may be read from and written to by the `read` and `write` primitives. Although the `write` primitive can be used anywhere, the `read` primitive can only be used within the arguments of a self-adjusting function call. A *self-adjusting function* is one that reads from one or more modifiables. These functions are defined in the usual way, but are invoked with the `call` primitive. Self-adjusting functions may invoke non-self-adjusting functions using ordinary C code.

The keys provided to `new` and `modref` support *indexed allocation*. In particular, during change propagation, a location or a modifiable may be reused if the given keys match, and if a few other conditions are met. The reuse mechanism guarantees correctness but may improve efficiency of change propagation by maximizing computation reuse. We discuss the reuse mechanism more in Section 3.

2.2 Program requirements

We assume that self-adjusting programs written with our interface observe the requirements described below. Our library currently makes no effort to check or enforce these. As such, the provided interface will probably serve best as an intermediate language for a high-level language providing static and/or dynamic enforcement of these requirements.

Self-adjusting functions must return `void`. Furthermore, for both self-adjusting functions and any function they invoke, we require the following:

1. Allocated modifiables are written exactly once and aren't read-from before they are written.
2. Blocks allocated with `new` aren't modified after initialization.
3. Except for modifiables, all accessed memory contains fixed values.

These constraints are motivated by (1) a need to record self-adjusting function calls and their arguments for later re-execution and (2) to accurately determine when the arguments and behavior of a these functions are affected. These requirements ensure that any change to the run-time behavior of a function is completely determined by its arguments. By restricting self-adjusting functions to return `void`, we force these functions to return values through modifiables. This ensures that all data dependences can be tracked by tracking the operations on modifiables.

2.3 An Example

As an example, Figure 3 shows the code for quicksort. The function is a recursive list-based implementation where the tail of each list cell is a modifiable and where the functions have been transformed into destination-passing style, returning results through modifiable arguments rather than through `return`. The function takes an unsorted input list l as well as a sorted list *rest* which is effectively appended to the result of sorting l. The result will be written to the destination d.

If l is empty, represented here by `NULL`, then the programmer writes *rest* to the modifiable d. Otherwise, they use the first element of l as a pivot and split l into two sublists as usual. They allocate destinations for the sublists by using the pivot as a key. Then, they recursively sort the two sublists. Using *pivot* for keying the allocations enables reusing the modifiables created for each pivot during change propagation. In previous work, it has been shown that this suffices to obtain efficient change propagation for quicksort [2]. In general, keying the allocations performed in the body of the function by the values used in the body suffices to obtain good performance. In this quicksort example, the output list is updated in expected $O(\log n)$ time for a random insertion or deletion in the input list.

2.4 Meta primitives

In self-adjusting computation, after the programmer executes a computation, at the *meta-level*, s/he can change the input to the computation and update the result by performing change propagation. The primitives described earlier are all available at the meta-level. For example, `write` may be used to change the input. We also provide the following primitives that can be used only at the meta-level:

- `deref`(l)**:** Returns the contents of the modifiable at l.
- `kill`(l)**:** Marks the allocation at l dead. This primitive can only be applied to a location l that is allocated at the meta level (all other dead locations will be found automatically).

```
typedef struct { int head; modref_t* tail; } cell_t;

void cell_init(void* block, void** keys) {
  cell_t* cell = (cell_t* ) block;
  cell->head = (int) keys[0];
  cell->tail = modref();
}

void split(cell_t* l, modref_t* d1, modref_t* d2, int pivot) {
  if (l == NULL) {
    write(d1, NULL); write(d2, NULL);
  } else {
    cell_t* cell = new(sizeof(cell_t), cell_init, l->head);
    if (l->head < pivot) {
      write(d1, cell);
      call(split, read(l->tail), cell->tail, d2, pivot);
    } else {
      write(d2, cell);
      call(split, read(l->tail), d1, cell->tail, pivot);
}}}

void quicksort(cell_t* l, cell_t* rest, modref_t* d) {
  if (l == NULL) {
    write(d, rest);
  } else {
    int pivot = l->head;
    modref_t* less = modref(pivot, 0);
    modref_t* grtr = modref(pivot, 1);
    call(split, read(l->tail), less, grtr, pivot);

    cell_t* pivot_cell = new(sizeof(cell_t), cell_init, pivot);
    call(quicksort, read(less), pivot_cell, d);
    call(quicksort, read(grtr), rest, pivot_cell->tail);
}}
modref_t* sorted = modref();
call(quicksort, read(unsorted), NULL, sorted);
```

Figure 3. Self-adjusting `quicksort`

```
cell_t* new_cell = new(sizeof(cell_t), cell_init, 23);
write(cell->tail, deref(unsorted));
write(unsorted, new_cell);
propagate();

write(unsorted, deref(new_cell->tail));
kill(new_cell->tail);
kill(new_cell);
propagate();
```

Figure 4. Change propagation with quicksort.

- `propagate()`: Updates the computation's output to account for any changes performed at the meta-level via `write`.

Figure 4 shows an example, where the programmer begins by sorting a list using `quicksort`. Next, they insert an additional element at the head of the input list by allocating a new cell holding 23 and perform change propagation by calling `propagate`. This updates the output by inserting 23 at the right position in the sorted list. Next, they remove the inserted cell from the input, mark the cell dead, and propagate again. This updates the sorted list by removing 23. It's important to note that the programmer need only mark locations as dead when they are allocated at the meta level, and never within the self-adjusting computation itself: our system will collect all locations allocated by the self-adjusting computation when and if they become unreachable during change propagation. In fact, as we describe in Section 3, this slows change propagation down only by a constant factor.

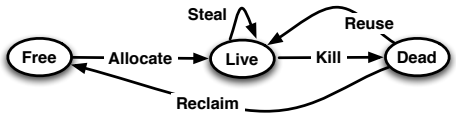

Figure 5. The memory model.

3. Algorithms and Implementation

This section describes the algorithms and data structures for tracing, change propagation, and garbage collection.

3.1 Overview

At a high-level, self-adjusting computation works by constructing a execution trace in a from-scratch run and using this trace for updating the output when the input changes.

In a from-scratch run, each invocation of a self-adjusting function is recorded by constructing a *closure* for the invocation and inserting it in the trace. The trace can be thought of as a sequence of such closures ordered by execution time. A closure records the function and the arguments of the invocation, the modifiables it reads from, and allocations it *owns*. A closure owns an allocation if it is performed within its immediate dynamic scope (i.e., its full dynamic scope minus the dynamic scope of its callees). This assigns a unique owner to each allocation.

After the from-scratch run is complete (and its corresponding trace is constructed), the user can then change one or more modifiables by writing them with new values and run change propagation. Change propagation updates the program's output and its trace. Change propagation processes the trace by performing one of three actions on each closure: (1) ignore, (2) remove or (3) re-execute.

To remove a closure, we delete it from the trace and place the locations it owned into a list of dead locations. To re-execute a closure, we move the locations it owned into the list of dead locations and re-execute the function invocation. When re-executed, the function may perform allocations, which we record in its closure. During change propagation, an allocation operation may reuse a dead location via indexed allocation.

At the end of change propagation, we reclaim the memory blocks at dead locations. We postpone reclaiming dead locations until the end of change propagation for two reasons. First, this enables reuse of dead locations. Second, a dead location may remain reachable in the trace until change propagation completes. By moving locations into the dead list when their owners are deleted, we essentially treat all such location as potential garbage. However, the uses of these locations (i.e., other blocks containing them) may still be live until the end of change propagation. [2] In these cases, reclaiming these dead locations eagerly would effectively create dangling pointers in the trace.

Figure 5 illustrates how the state of memory locations may transition both during from-scratch execution and during change propagation. Initially, all locations are *free*. When a closure allocates a location, this location is then considered *live*. When a closure is removed from the trace, all allocations it formerly owned are *killed* and considered *dead*. During change propagation, a closure c_1 being reevaluated can potentially *steal* an indexed *live* location from another closure c_2 by supplying matching keys. Additionally, c_1 may *reuse* a formerly *dead* location indexed by matching keys. When change propagation is complete, each *dead* location corresponds to a memory allocation that is no longer performed. Such locations are therefore unreachable through the trace and can safely be reclaimed. This is because after change propagation, the trace is

[2] It is a property of change propagation that these pointers will not be followed, and will only be used for equality tests.

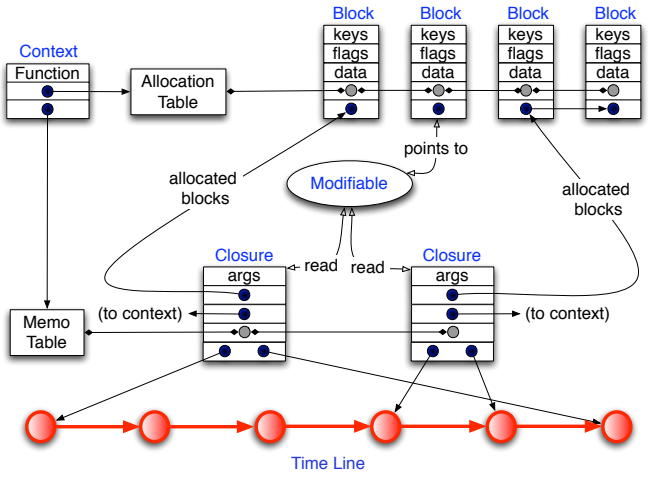

Figure 6. Internal data structures.

consistent with a from-scratch run, and any allocations no longer taking place cannot be used within this run.

3.2 Data Structures

We describe the data structures used for representing various run-time objects. Figure 6 shows a snapshot of these data structures during a hypothetical execution.

Order Maintenance Data Structure. An *order-maintenance* data structure maintains a set of *time-stamps* while supporting the following operations in constant time: create and insert a new time-stamp after another, delete a time-stamp, and compare two time-stamps [16]. The implementation uses an order-maintenance data structure to represent the *time-line* of the execution. In the figure, the time-line is the linked listed of time-stamps at the bottom.

Allocation and Memo Tables. An allocation table is a hash table that maps a set of keys to memory blocks. A memo table is a hash table that maps sets of arguments to closures. These tables are used to reuse allocations and computations (by reusing closures) during change propagation.

Contexts. All functions execute with respect to a *context* that is shared by a function and all its recursive invocations. We represent the context as a record that consists of a function pointer, an allocation table, and a memo table.

Closures. Each self-adjusting function call creates a closure that consists of a pointer to its context, arguments, pointers to any modifiables specified for reading by the arguments, its entry in the memo table, pointers to its start and end time-stamps and pointers to memory blocks allocated during the execution of the closure (but not including those allocated by nested calls).

Modifiables. A modifiable is either empty or written. If written then it has a value and a *reader list* that consists of all closures that read it. The value has the size of a machine-word and holds either a small value (e.g., an integer) or a pointer to a memory block.

Blocks. Each memory allocation creates a memory block that consists of the keys used for that allocation, some flags used for memory management purposes and space for data of the specified size. A block created by a keyed allocation is stored in an allocation table to which the block holds pointers. We refer to the closure that allocates a memory block as the *owner* of the block. All blocks allocated by a closure are chained together as a doubly linked list and are accessible given the closure. These pointers help identify memory blocks for reuse and reclamation.

In the hypothetical example shown in Figure 6, we have two closures that are recursive invocations of the same function call.

```
function deleteComputation (t1, t2) =
  for t in (t1, t2)
    remove t from time-line
    if (t = c.start) for some c then
      remove c from its context's memo table
      remove c from any read-lists
      move allocations of c into dead-list

function call (f, a) =
  if not a recursive call then
    current-context ← new context

  c ← find (current-context.memo-table, a))
  if (c ≠ null) then
    deleteComputation (current-time,c.start)
    propagateInt(c.start, c.end)
  else
    t1 ← advance-time ()
    c ← make closure for f with a
    current-closure ← c
    run c
    t2 ← advance-time ()
    c.start ← t1
    c.end ← t2
    insert c into current-context.memo-table
    restore current-context
    restore current-closure

function propagateInt(t1, t2) =
  while (|Q| > 0 and top(Q).start in (t1, t2))
    c ← pop(Q)
    current-time ← c.start
    current-context ← c.context
    current-closure ← c
    reuse-end-time ← c.end
    refresh c
    move allocations of c into dead-list
    run c
    deleteComputation(current-time, end(c))

    recover current-context, current-closure,
      reuse-end-time

function propagate () =
  propagateInt (time-line.start, time-line.end)
  reclaim all locations in dead-list
```

Figure 7. Pseudo-code for function calls and change propagation.

The closures therefore point to the same context. The first closure calls the second closure and therefore the execution time interval of the second is contained within that of the first. Both closures read a modifiable and each allocate memory blocks. The modifiable has been written with the location of a fourth block. The memory blocks are all stored in the allocation table and the blocks allocated by each closure are linked together. To simplify the figure, we omit some details such as the buckets of the allocation and memo tables, the read-lists for modifiables and some back pointers between data structures.

3.3 The Primitives

We describe the algorithms for the primitives of our library. Figure 7 shows the pseudo code for some of the algorithms.

During execution we maintain a *time-line* that is initialized with two time-stamps marking the beginning and the end of the time-line and remember the `current-time` (initialized to be the beginning time of the time-line). We often *advance time* by inserting a new time-stamp t immediately after `current-time` and setting the

current-time to t. We maintain a *reuse-interval* as a pair of time-stamps consisting of the current-time and reuse-end-time that mark the beginning and the end of the time interval in which reuse can take place. All memo-table and allocation-table lookups return a match only if the specified keys match the keys of an entry that falls within the reuse interval in the time-line.

For memory management, we maintain a list of dead locations; the live and free locations are represented implicitly.

The time-line effectively represents the execution time-line, and guides change-propagation by helping identify which closures and memory locations can be reused and which will become garbage. During change propagation, we remove closures from the time-line and mark all the allocations performed by them dead by using a deleteComputation function. Given an interval of two time stamps t_1 and t_2, we delete the computation between these two time-stamps by deleting each time-stamp in this interval, including the closure that starts at each time-stamp, if any. To delete a closure, we mark dead all memory allocated by that closure by inserting them to the dead list. We run deleteComputation at two points during change propagation: when a memo match takes place and after a closure is re-executed.

To reuse computations and match keys, we rely on physical equality tests. This means that two memory objects are considered equal if they have the same memory address.

For storing the closures that are affected by the changes, we keep a priority queue Q which we initialize to empty. In addition, we also keep a pointer called current-closure to the closure being currently executed and a pointer current-context to its context.

For the new and modref primitives, we identify a location l to return based on the keys given as follows. If no keys are provided we allocate a fresh location l for an appropriately-sized block from the free set. If some keys are specified, then we first consult the allocation table of the current-context. If we find a match to some location l, then we check if it has an owner. If l has an owner c, then it is live and we perform a steal by inserting c into Q, removing l from c's allocation list, and inserting l into the allocation list of the current-closure. If l has no owner, then l is in the dead list. In this case we perform a reuse by moving l from the dead list into the allocation list of current-closure. If no match was found for the given keys, we allocate a fresh location l and insert it into the allocation table of the current-context. After we identify l, we insert it into the allocation list of the current-closure.

The primitives that make up our interface from Section 2 are described below.

new First, we choose l based on the keys given. If l is fresh, we call the given initialization function with l and the given keys.

modref First, we choose l based on the keys given. If l is fresh we initialize the modifiable to be empty (unwritten) with an empty reader-list. If l is reused or stolen, then we keep both its value and list of readers.

write Given a modifiable and a value v, we write an empty modifiable by placing v as it's value. To write a written modifiable we compare the value it holds v_0 with v. If they are equal, we return. If they differ, we insert each closure c in the reader-list of the modifiable into the affected queue Q, clear the reader-list, and return.

call To call a function f with arguments a we check if the f is a recursive invocation of the function currently being executed. If not, then we create a new context; otherwise we continue using the existing context. We then check if there is a closure that matches the call by checking the memo-table. If there is a match then we delete the computation from the current-time to the start time of the matched closure and perform change propagation on the reused closure. This adjusts the reused computation to changes. If there is no match, then we advance the current time to t_1, create a closure for f with the given arguments, set the current-closure to the closure, and run the closure. After the closure returns, we advance the time to t_2 and make (t_1, t_2) the time interval for the closure. We store the closure in the memo table, restore the current-context and current-closure and return.

propagate ***and*** *propagateInt* Given a set of affected closures and a time interval (t_1, t_2) where it will be applied, change propagation re-executes the affected closures from Q in the order of their start times. To re-execute a closure c, we first set the current-time, current-context, and current-closure appropriately. We then refresh the closure by reading the modifiables in its arguments, mark the allocations of c garbage, and perform the call. After the call completes, we delete the computation between the current-time and the end time of the closure. Change propagation completes when there are no more affected closures to execute within the specified time interval (t_1, t_2).

3.4 Meta Level

It is reasonably straightforward to support the meta-level primitives by using the primitives described in Section 3.3. The deref primitive simply returns the contents of a modifiable. The kill primitive marks the specified location dead by inserting it into the dead-list. The pseudo-code code for propagate is shown in Figure 7. It calls propagateInt from the beginning of the time-line to its end by specifying the first and the last time stamps of the time-line. After propagateInt finishes all the locations marked as dead are reclaimed by removing them from their allocation tables and freeing the blocks that they point to.

3.5 Performance

We show that our algorithms for tracing, change propagation, and garbage collection are efficient. For the proof we make two assumptions about the use of allocation primitives: (1) all indexed allocations are keyed uniquely, (2) all function calls have unique arguments. These assumptions ensure that hash-table accesses take constant time. We note that these assumptions cause no loss of generality: the run-time system can extend user-provided keys or arguments with an additional counter to distinguish between equal keys or arguments. All our bounds are randomized with expectations taken over internal randomization used for representing hash tables (used for memoization and indexed allocation).

The described systems satisfy the following properties:

1. In a from-scratch run, the overhead of tracing is expected $O(1)$.
2. During change propagation, the time per reclaimed location is $O(1)$.

For space reasons we omit the full proof, but provide a sketch here. The first property follows by inspection of the primitives from Section 3.3. The new, modref, write and call operations each require expected $O(1)$ time because in a from-scratch run there is no reuse of allocations or closures. For the second property, note that a reclaimed location must be inserted into the dead list. Thus the total time for reclamation is bounded by the size of the list. Since each reclaimed location is inserted into the dead list once and since there are no further operations on dead locations, it follows that the time per reclaimed location is $O(1)$. We verify these theorems empirically in Section 5.

3.6 Implementation

We implemented the proposed approach in C. Our implementation is available online at:

http://ttic.uchicago.edu/~hammer/ismm08/

We discuss a few additional details of the implementation next.

The free set. The free set described in Section 3.1 is implemented as a standard one-level allocator that maintains a collection of free lists, each of which is used for allocating objects of a fixed size [23]. We extend a free list by allocating one page and dividing it into blocks of the desired size. For objects larger than one page, we use `malloc` directly.

It is well-known that fragmentation can occur in non-moving collectors such as our collector. There are several proposed solutions to this problem [23]. A more comprehensive study of fragmentation and which of the previously proposed approaches may perform best in self-adjusting applications is beyond the scope of this paper. We note, however, there are several relatively simple approaches that can be effective in our setting. One such approach is to re-executing the whole program from scratch to compactify. Since the trace data (and thus the live data) is often in the same order as the running time of a program, this can work well, especially when it is used rarely. More interestingly, instead or re-executing from scratch to compactify, we can compact incrementally by re-executing only a "tail" of a computation, i.e., the rest of the computation after a particular execution time. This is achieved by re-executing the function invocations that start after a particular time—such functions can be identified by scanning the time-line. Re-executing a tail suffices because it does not require moving data allocated prior the start time of the tail in the execution.

Tail recursion. Many of the tail-recursive functions we implement are transformed by our library's C macros into ordinary loops, where the looping occurs at tail-recursive uses of the `call` primitive. To support this, we also modify the creation of these closures' end times, so that a series of tail-recursive calls share a single time-stamp as their end time. We store this end time in the context of the recursive sequence, which is already shared by each closure in this sequence. By using this optimization, tail-recursive functions don't grow the program's stack, which is ordinarily necessary in order to assign end times to their closures.

4. Applications

To evaluate our C implementation we implemented a number of applications, some of which we directly translated from the previous work on self-adjusting computation done with Standard ML [4]. We use these applications as a basis for comparison between our implementation and the previous implementation and for measuring the effectiveness of our memory management techniques. In addition, we include several additional applications not considered in the previous work. Broadly, we classify the applications into list primitives, sorting applications, computational-geometry applications, and tree applications.

List primitives and Sorting. Our list applications are `filter`, `map`, `minimum`, and `sum`: `filter` takes a list and a predicate and returns a list of elements that satisfy that predicate; `map` takes a list and a function and returns another list by applying the function to each element of the list; `minimum` takes a list and returns the minimum element in the list; `sum` takes a list and returns the sum of the elements in the list. Our sorting applications, `quicksort` and `mergesort`, are implementations of the classic sorting algorithms of the same name. These benchmarks are directly taken from previous work and implemented in the same way [4].

Computational Geometry. Our computational-geometry applications are `quickhull`, `diameter`, and `distance`: `quickhull` and `diameter` are implementations of two classic algorithms for finding the convex hull and the diameter of points in the plane [10]. Their implementations are identical to those in the previous work. The `distance` application computes the distance between two point sets, by first finding their convex hulls (using `quickhull`) and then finding the minimum distance between the points in the convex hull. Computation of distances between points sets or convex objects is critical in collision detection.

Tree Applications. Our tree applications are `exprtree` and `bstverif`. The `exprtree` application takes expression trees e of the form $e ::= e \oplus e \mid z \mid \mathsf{error}$, where $\oplus \in \{+, -, /, \times\}$ and $z \in \mathbb{Z}$, and either computes the value of the tree or yields error if the expression contains a divide-by-zero. The "input" to `bstverif` is a binary search tree undergoing imperative updates (e.g., insertions and removals of keys). The application verifies the invariants of the tree recursively, checking that each key is properly ordered with respect to any parents or children, and that for each node the parent pointer actually points to the node's parent. Unlike the other applications, `bstverif` traverses a cyclic data structure (a tree with parent-pointers) and doesn't produce output. It sends the abort signal (`SIGABRT`) to it's process if an invariant has been violated. This application shows an example of incremental data structure verification in our system. We expect that other commonly-used C data-structures can be incrementally verified in a similar way.

5. Experiments

We present an experimental evaluation of our implementation on a set of benchmarks derived from our applications (Section 4).

Benchmarks. Each benchmark is derived from an application (usually with the same name) described in Section 4. The input to our list benchmarks are integer lists. The `filter` benchmark filters out even elements. The `map` benchmark adds a fixed value to each value in the list. The remainder of the applications are translated into benchmarks of the same name as one would expect.

For each benchmark we consider two C versions which we call *static* and *self-adjusting*, respectively. Each static version is derived from a corresponding self-adjusting version by replacing modifiables with ordinary reference cells and omitting the bookkeeping required for self-adjustment. The static version essentially eliminates the overheads for performing dependence tracking and change propagation (e.g., time stamps, memo tables, dependence graphs). Moreover all memory allocation operations are directed to use C's allocator `malloc` skipping the overheads associated with our memory allocator (e.g., allocation tables, allocation lists). By comparing the from-scratch runs of the static and self-adjusting versions we report an estimate of the overhead associated with our system. By comparing the from-scratch run of the static version to the time required for change propagation in the self-adjusting version, we evaluate the speedup delivered by self-adjustment. [3]

Many of the benchmarks also have self-adjusting versions in SML, which we report including and excluding the cost of garbage collection performed by these versions.

Input generation. For each benchmark we generate random input. We denote the size of a benchmark's input as n, and we vary n to demonstrate asymptotic behavior. For sorting and list-processing benchmarks, the input lists consist of n integers chosen randomly. For `quickhull` and `distance`, n points are drawn from a uniform distribution over the unit square in $\mathbb{R}^2$. For `distance`, two non-overlapping unit squares are used, and from each square we draw $n/2$ points. For the `exprtree` benchmark, we generate random binary trees with n nodes and expected $\log n$ depth. For each internal node one operation $\oplus$ is chosen at random, and at each leaf a random integer z is placed. For the `bstverif` benchmark, we first build a BST of size n, and then perform n removals and insertions to the tree. For a tree of size n we use keys $A = \{1, ..., n\}$ and in-

[3] We also experimented with versions of our static benchmarks that use our allocator (Section 3.6) instead of `malloc` and found that it is up to a factor of two faster depending on the application.

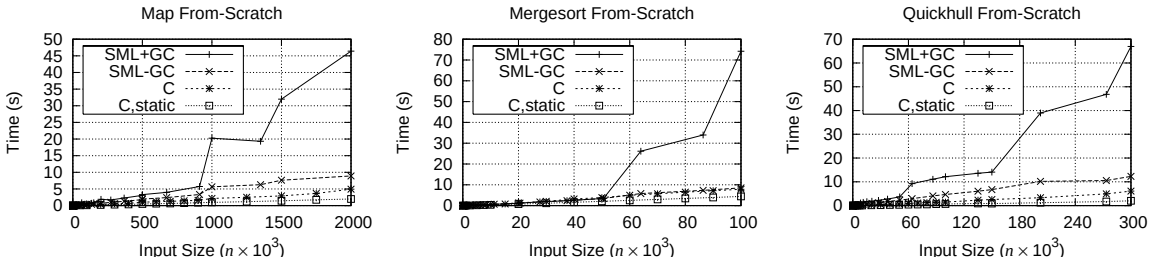

Figure 8. From-scratch runs for `map`, `mergesort` and `quickhull`

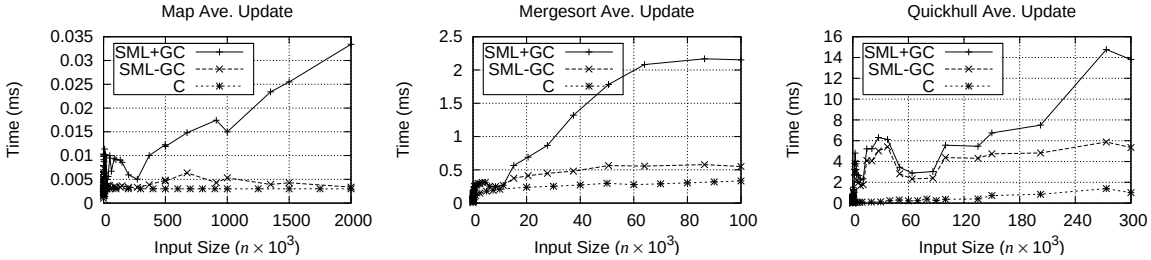

Figure 9. Average update times for `map`, `mergesort` and `quickhull`

sert them into an initially-empty tree in a random order. In a second random order, for each key $a \in A$, we remove and then re-insert a. For each operation, the BST data-structure incrementally runs the `bstverif` application to verify that the tree satisfies its invariants.

5.1 Measurements

The measurements we present were gathered on a dual 2Ghz PowerPc G5 with 6 gigabytes of memory and running Mac OS X. All times are wall-clock times and are measured in seconds. For the SML timings, we compile via MLton with the flags "`-runtime "ram-slop 1.0"`", which instructs the MLton runtime to expand the heap aggressively when needed[4]. For the C timings, we do whole-program compilation using GCC version 4.0.2 and the flags "`-O3 -combine`".

For each benchmark, we run both the static and self-adjusting versions on input of size n and record their running times. For each self-adjusting version, we perform a more detailed timing test. The timing test consists of the from-scratch run of the self-adjusting version followed by updates throughout the input of the program. At each position of the program's input we perform two changes: an insertion of a new input element (generated in the same fashion as the original input elements) and the removal of this element. We follow each of these with a run of the change-propagation algorithm. We call each change and it's propagation an *update*. Several of our measurements are given as averages over all possible updates for a given benchmark on a given input size. We also include memory measurements that consider the maximum amount of live memory at any point during the timing test.

5.2 Time analysis: map, mergesort, quickhull

We give detailed results for three of our benchmarks: `map`, `mergesort`, and `quickhull`. We summarize all the benchmarks in Section 5.3.

Figure 8 plots the time of the from-scratch runs and Figure 9 plots the average time of change propagation after an insertion or deletion. We consider the running time for the self-adjusting (labeled "C") and static versions (labeled "C,static") of our benchmarks. For comparison, we also include timings for the SML implementation and report them both including and excluding GC time (labeled "SML+GC" and "SML-GC" respectively).

Comparing the static and self-adjusting timings in C shows that:

- The self-adjusting versions in C are a small constant factor slower than the static versions in C. This constant is about 2.
- The time for change propagation in C either remains bounded by a constant regardless of the input size or grows logarithmically in the input size. These results are consistent with the best possible update times for these benchmarks ($O(1)$, $O(\log n)$ and $O(\log n)$ for `map`, `mergesort` and `quickhull` respectively).
- The SML versions of the benchmarks spend a significant percentage of their running time performing GC, especially as the input sizes increase. This can be observed by comparing the "SML-GC" and "SML+GC" measurements. Even though "SML-GC" and "C" timings are quite close, when the GC times are included ("SML+GC") the gap between our C version and the SML versions can increase more than ten fold. This confirms that our approach dramatically reduces the memory-management overheads from previous work.

5.3 Time analysis summary

Table 1 summarizes our timing measurements for all of our benchmarks at fixed input sizes (note that for tree applications and `distance` we have no corresponding SML versions).

The measurements show that the conclusion drawn from the data for `map`, `mergesort` and `quickhull` generalize to other applications. Our C benchmarks are reasonably close to the SML benchmarks when the GC time is excluded ("SML-GC"). However, when the GC times are included in the SML timings ("SML+GC"), the C benchmarks are up to an order of magnitude faster for both from-scratch runs and change-propagation. Across all the benchmarks the C versions are more than a factor for 5 faster on average even though they are only a factor of 2 faster when the GC times in SML are excluded. We note that the input sizes used in the timings are relatively modest in the sense that the space usage of the SML benchmarks do not exceed half of the available physical memory. As we discussed previously (Section 5.2), the gap between SML and C increases further as the input sizes increase.

Table 2 reports the overheads and speedups for all our benchmarks. As can be observed, all our overheads are less than a factor of 4 (the average is 2.3). The speedups vary between three and five orders of magnitudes.

[4] We find this gives comparable results to running with a large fixed-size heap.

		From-Scratch				Propagation			Space	
Application	n	C,static	C	SML+GC	SML-GC	C	SML+GC	SML-GC	C	SML
`filter`	10^6	0.5	2.1	6.3	3.9	2.7×10^{-6}	2.0×10^{-5}	6.7×10^{-6}	137.0	558.3
`map`	10^6	0.9	2.1	17.7	5.3	3.0×10^{-6}	2.5×10^{-5}	4.6×10^{-6}	152.6	663.4
`minimum`	10^6	1.2	3.1	7.6	5.0	9.0×10^{-6}	3.4×10^{-5}	1.7×10^{-5}	167.3	509.9
`sum`	10^6	1.3	3.1	7.8	4.5	8.8×10^{-5}	1.1×10^{-4}	8.4×10^{-5}	167.2	454.0
`quicksort`	10^5	2.3	4.9	21.1	5.4	4.1×10^{-4}	1.8×10^{-3}	4.1×10^{-4}	353.7	654.4
`mergesort`	10^5	4.4	7.9	74.2	8.5	3.3×10^{-4}	2.2×10^{-3}	5.5×10^{-4}	676.4	1143.1
`quickhull`	10^5	0.7	1.5	12.3	4.7	3.7×10^{-4}	5.6×10^{-3}	4.4×10^{-3}	113.5	347.9
`diameter`	10^5	0.7	1.6	9.4	4.4	3.6×10^{-4}	3.0×10^{-3}	1.8×10^{-3}	114.1	337.9
`distance`	10^5	0.7	1.4	*n/a*	*n/a*	2.0×10^{-4}	*n/a*	*n/a*	107.1	*n/a*
`exprtree`	10^6	1.5	3.4	*n/a*	*n/a*	1.5×10^{-4}	*n/a*	*n/a*	328.5	*n/a*
`bstverif`	10^6	2.4	9.4	*n/a*	*n/a*	2.0×10^{-5}	*n/a*	*n/a*	486.0	*n/a*

Table 1. From-scratch and propagation times (in seconds) and space (in megabytes) for the benchmarks.

Application	n	Overhead	Speedup
`filter`	10^6	4.2	1.7×10^5
`map`	10^6	2.4	3.0×10^5
`minimum`	10^6	2.6	1.3×10^5
`sum`	10^6	2.4	1.5×10^4
`quicksort`	10^5	2.1	5.6×10^3
`mergesort`	10^5	1.8	1.3×10^4
`quickhull`	10^5	2.1	1.9×10^3
`diameter`	10^5	2.3	1.9×10^3
`distance`	10^5	2.0	3.5×10^3
`exprtree`	10^6	2.3	1.0×10^4
`bstverif`	10^6	3.9	1.2×10^5

Table 2. Overhead and speedup in C.

5.4 Space analysis

Table 1 reports the maximum amount of live memory for each benchmark for both the C and SML versions. In all cases, this space measurement considers all live data, including tracing structures used internally (e.g., time stamps, memo tables, dependence graphs, *etc.*). As can be observed, the C versions consistently use less space than the corresponding SML versions (up to 75% less).

We suspect this results from differences between the trace representations of the C and SML libraries. For example, in C we can often associate several reads with a single closure, whereas in SML each read is represented explicitly. Additionally, in C we are free to use physical addresses for equality and hashing, whereas the SML library uses unique tags for changeable data, introducing a further level of indirection.

5.5 Summary

Our experiments confirm that our approach to memory-management for self-adjusting computation is effective in practice. We show that the overheads are reasonably small in comparison to static versions (about a factor of two on average) and that our speedups are several orders of magnitude, depending on the application. These results confirm our theoretical claims. Compared to the previous implementations in SML, we observe up to an order of magnitude reduction in running times, which we confirm is due to GC by reporting and comparing timings including and excluding GC time.

6. Related Work

Incremental and Self-Adjusting Computation. The problem of enabling computations to respond to changes automatically has been studied extensively. Most of the early work took place under the title of *incremental computation* (e.g. [15, 31, 18, 34, 39, 1, 24, 21]). Here we briefly review some of the work most related to our approach and refer the reader to the bibliography of Ramalingam and Reps [32] for a more detailed set of references.

Perhaps the two most effective approaches to incremental computation are based on static dependence graphs [15] and memoization [31, 27]. Static dependence graphs represent the dependences in a computation as a graph that can be used to propagate changes to computation data. During propagation, static dependence graphs do not permit the structure of the dependences to change; this limits their applicability to a certain domain of applications. An alternative proposal [30, 31] is based on the classic idea of (function) memoization [11, 26, 27]. Unfortunately, with memoization, it is generally difficult to update in optimal time. Self-adjusting computation generalized dependence graphs by introducing dynamic dependence graphs [5] and showed that there is an interesting duality between dynamic dependence graphs and memoization [4]. The change-propagation algorithm used in self-adjusting computation exploits this duality to perform efficient updates.

Garbage Collection. Garbage collection has been shown to be effective in improving programmer productivity often without decreasing performance significantly (e.g.,[40, 12]). Numerous techniques for garbage collection (e.g., [25, 14, 9, 17, 36]) have been proposed and studied (e.g., [13, 38] for surveys). Traversing collectors have been extended to offer domain-specific customization, such as *simplifiers* [28]. Simplifiers can help reduce total live memory by running programmer-supplied code during memory traversal. In contrast to all of these approaches, we perform memory management without traversing memory. In this sense, our approach is similar to reference-counting collectors. However, our approach does not require maintaining reference counts and is able to deal with cyclic data structures directly.

Perhaps the most closely related work to ours is the work on region-based memory management (e.g., [36, 19, 20, 35]). Although we do not use region-based memory management techniques, it might be possible to implement our approach based on them. To do this, we would assign a region to each function invocation holding the blocks allocated by that invocation. We would then rely on a region inference algorithm or use a region type system to establish the liveness conditions of our regions. In particular, either the inference algorithm or implementor would have to prove that a region can be treated as garbage when the function invocation that it belongs to is deleted from the execution trace. However, this seems difficult because objects residing in regions remain reachable until the end of change propagation. In addition to this problem, it also seems difficult to implement reuse of allocations with regions—to do this we would need the ability to move objects between regions. Instead of relying on region inference or a region type system, we use meta-level reasoning to determine when a region becomes garbage. In particular, when change propagation removes a function invocation from the execution trace, we know that the region for that invocation will become garbage at the end of

change propagation—this holds because change propagation yields a computation that is consistent with a from-scratch execution.

7. Conclusion

Previous work showed that memory management overheads can be very high in self-adjusting computation and can even cause the approach to scale poorly to large inputs. In this paper, we develop memory-management techniques that reduce memory management overheads by providing an efficient memory allocator and a garbage collection algorithm that reclaims garbage incrementally—the approach requires $O(1)$ time for each reclaimed memory block. This is achieved by combining garbage collection and change propagation, the mechanism with which self-adjusting programs respond to changes. The correctness of the approach relies on the consistency or contextual equivalence property of change propagation which states that change propagation and from-scratch executions are equivalent. We implement the approach in C and perform an experimental evaluation. Our experiments show that the approach is very effective in practice. We observe small overheads and high, scalable speedups. Compared to previous implementations of self-adjusting computation, we observe up to a factor of 10 improvement in timings, while reducing space usage up to 75%.

References

[1] M. Abadi, B. W. Lampson, and J.-J. Levy. Analysis and caching of dependencies. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 83–91, 1996.

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

[3] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, 2008.

[4] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[5] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2006.

[6] U. A. Acar, G. E. Blelloch, and K. Tangwongsan. Kinetic 3d convex hulls via self-adjusting computation (an illustration). In *Proceedings of the 23rd ACM Symposium on Computational Geometry (SCG)*, 2007.

[7] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and J. L. Vittes. Kinetic algorithms via self-adjusting computation. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pages 636–647, September 2006.

[8] U. A. Acar, A. Ihler, R. Mettu, and O. Sümer. Adaptive bayesian inference. In *Neural Information Systems (NIPS)*, 2007.

[9] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[10] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[11] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[12] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, Nov. 2002. ACM Press.

[13] J. Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, Sept. 1981.

[14] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[15] 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.

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

[17] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory subsystem performance of programs with intensive heap allocation. Technical Report CMU-CS-93-227, Computer Science Department, Carnegie-Mellon University, Dec. 1993. Also appears as Fox Memorandum CMU-CS-FOX-93-07.

[18] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LISP and Functional Programming*, pages 307–322, June 1990.

[19] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In PLDI [29], pages 282–293.

[20] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In PLDI [29], pages 141–152.

[21] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 311–320, 2000.

[22] B. Hudson. *Dynamic Mesh Refinement*. PhD thesis, Carnegie Mellon University Computer Science Department, Dec. 2007.

[23] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[24] Y. A. Liu, S. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

[25] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[26] J. McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[27] D. Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, 1968.

[28] M. E. O'Neill and F. W. Burton. Smarter garbage collection with simplifiers. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 19–30, New York, NY, USA, 2006. ACM.

[29] *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[30] W. Pugh. *Incremental computation via function caching*. PhD thesis, Department of Computer Science, Cornell University, August 1988.

[31] 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.

[32] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 502–510, 1993.

[33] A. Shankar and R. Bodik. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming language Design and Implementation (PLDI)*, 2007.

[34] R. S. Sundaresh and P. Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–13, 1991.

[35] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3), Sept. 2004.

[36] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, Feb. 1997.

[37] S. Weeks. Whole-program compilation in mlton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM.

[38] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 Sept. 1992. Springer-Verlag.

[39] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, Apr. 1991.

[40] B. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23:733–756, 1993.