Two for the Price of One: A Model for Parallel and Incremental Computation

Sebastian Burckhardt
Daan Leijen
Microsoft Research
{sburckha,daan}@microsoft.com

Caitlin Sadowski
Jaeheon Yi
Univ. of California at Santa Cruz
{supertri,jaeheon}@cs.ucsc.edu

Thomas Ball
Microsoft Research
tball@microsoft.com

Abstract

Parallel or incremental versions of an algorithm can significantly outperform their counterparts, but are often difficult to develop. Programming models that provide appropriate abstractions to decompose data and tasks can simplify parallelization. We show in this work that the same abstractions can enable *both* parallel and incremental execution.

We present a novel algorithm for parallel self-adjusting computation. This algorithm extends a deterministic parallel programming model (concurrent revisions) with support for recording and repeating computations. On record, we construct a dynamic dependence graph of the parallel computation. On repeat, we reexecute only parts whose dependencies have changed.

We implement and evaluate our idea by studying five example programs, including a realistic multi-pass CSS layout algorithm. We describe programming techniques that proved particularly useful to improve the performance of self-adjustment in practice. Our final results show significant speedups on all examples (up to 37x on an 8-core machine). These speedups are well beyond what can be achieved by parallelization alone, while requiring a comparable effort by the programmer.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Software*]: Programming Languages—Language Constructs and Features

General Terms Languages, Performance

Keywords Self-adjusting computation, Incremental memoization, Parallel programming

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10...$10.00

(a)		**(b)**

`while (true) {`
`  compute();`
`  mutate();`
`}`
	compute:
	- deterministic
	- potentially parallel
	- no I/O
	
	mutate:
	- nondeterministic
	- may perform I/O
		`record {`
		`  compute();`
		`}`
		`while (true)`
		`{`
		`  mutate();`
		`  repeat;`
		`}`

Figure 1. **(a)** Imperative compute-mutate loop. **(b)** An equivalent loop using our record and repeat commands that can exploit redundancy in the computation.

1. Introduction

Many programs perform repetitive tasks but are incapable of exploiting the redundancy inherent in the repeated computation. For example, in reactive or interactive programs, we often find *compute-mutate* loops as shown in Fig. 1(a). In such a loop, a deterministic computation, represented by compute, is repeated. Between repetitions, the user or environment may read and write the state in a nondeterministic manner, represented by the function mutate. The changes performed by mutate may be small or large, yet compute is always fully repeated, which is potentially wasteful.

For example, a web browser recomputes the layout of a page many times, whenever the page content changes based on user actions or the execution of scripts. If the changes are small, much of the layout computation is redundant; discovering and exploiting such redundancy manually or automatically is difficult in practice. Other examples of compute-mutate loops appear in games, compilers, spreadsheets, editors, forms, simulations, and so on.

Researchers have long recognized the performance potential of incremental computation. For many algorithms, specialized versions can adjust quickly to changes in the input, typically improving the asymptotic complexity. Moreover, many general techniques to incrementalize a computation automatically or semi-automatically have been proposed. For some algorithms, standard techniques like memoization or dynamic programming may be sufficient. For imperative programs with side-effects, researchers have proposed *self-adjusting computation*, which can track data and control dependencies, selectively repeating only the parts of the computation that depend on changed inputs.

Since the performance gains can be quite dramatic, self-adjusting computation is a viable alternative to parallelization (where it applies). Parallelization has been well studied as a means of improving the performance of applications, but its success often requires serious thought and effort by the programmer. Over the years, researchers developed many *programming models* (embodied by languages, language extensions, or libraries) that facilitate parallelization by providing suitable abstractions to the programmer.

Self-adjusting computation and parallelization exhibit some deep similarities: both aim to improve performance and both need to leverage the structure of the computation to do so successfully. This similarity in purpose and method is not merely of academic interest. In this paper, we demonstrate that it is possible and sensible to write programs that are *both parallel and incremental*, and can thus reap the performance benefits of both.

We show that a single, small set of programming primitives (record, repeat, fork, join, and type declarations) are sufficient to simultaneously exploit parallelism and perform self-adjusting computation, at a programming effort comparable to traditional parallelization. The crucial insight is that once the programmer declares how to divide a task into independent parts, both self-adjustment and parallelization can benefit.

Overall, we make the following contributions:

- We present a single, small set of primitives that allow programmers to simultaneously express the potential for parallelization and incrementalization in their applications.
- We describe an algorithm for parallel self-adjusting computation, i.e. an algorithm that can (1) record dependencies of a deterministic parallel computation, and (2) repeat the computation while only reexecuting parts whose dependencies have changed.
- We implement our model as an extension to the concurrent revisions system, deployed in the form of a C# library, and apply it to five sample applications written in C#, including a multi-pass CSS layout algorithm.
- We describe three programming techniques (granularity control, simple outputs, and markers) that enable self-adjusting computation to perform better in practice.
- We demonstrate that for all of the studied examples, self-adjustment can provide significant performance improvements compared to sequential or parallel execution only, while requiring a programming effort comparable to manual parallelization. Across our benchmarks we observe a 12x to 37x speedup compared to the sequential baseline on an 8-core machine.

Section 2 gives an overview of our programming model. In Section 3 we describe the algorithm and give a correctness argument. In Section 4 we discuss the benchmarks, the programming techniques, and the performance results, concluding in Section 5 with a discussion of related work.

2. Overview

In this section, we establish the parameters for our work. We begin with a simple illustration example that records a computation of a parallel sum, changes one of the summands, and then repeats the computation. We then describe the essentials of the concurrent revisions model and the various isolation types in more detail.

2.1 Programming Model

Our work builds on *concurrent revisions*, a recently proposed deterministic parallel programming model [14]. In this model, the programmer forks and joins tasks called *revisions* that can execute in parallel. The programmer must declare data that is shared by concurrent revisions to have an *isolation type*. The model then guarantees deterministic parallel execution [16] by copying memory locations that are accessed concurrently, and by resolving conflicts deterministically. We summarize the essentials of this model in Section 2.5.

To extend the concurrent revisions model with support for self-adjusting computation, we add record and repeat commands that let the programmer identify the computation that is to be repeated (Fig. 1 (b)). Moreover, we require that the programmer use isolation types not only for concurrently accessed variables, but also for variables that are accessed by consecutive revisions and that thus may introduce data dependencies.

2.2 Parallel Sum Example

The pseudo-code in the small example program in Fig. 2 demonstrates how to record and repeat a parallel summation using record and repeat commands. Summation follows a parallel divide-and-conquer structure and switches to sequential summation below some threshold, a common practice for parallel algorithms. When programming with concurrent revisions, all variables that may be accessed by concurrent revisions or by consecutive revisions must be declared using an *isolation type*.

In main(), we first allocate and initialize an array of versioned integers. Then we record the computation, a parallel summation using recursive divide-and-conquer and fork-join parallelism with a user-specified threshold for switching over to sequential summation. In ParallelSum(), the variable sum is declared to have type CumulativeInt. This data type declares sum as versioned (and isolated in each revision), and on a join operation sum combines any additions that were done in parallel. This is very similar in concept to hyperobjects in Cilk++ [22]. The first computed total is 1000. Next,

```
const int threshold := 250 ;

int ParallelSum(Versioned⟨int⟩[] a, int from, int to) {
  if (to − from ⩽ threshold)
    return SequentialSum(a, from, to) ;
  else {
    CumulativeInt sum := 0 ;
    Revision r1 := fork {
       sum := sum + ParallelSum(a, from, (from + to)/2) ;
    }
    Revision r2 := fork {
       sum := sum + ParallelSum(a, (from + to)/2, to) ;
    }
    join r2 ;
    join r1 ;
    return sum ;
  }
}

int SequentialSum(Versioned⟨int⟩[] a, int from, int to) {
    int sum := 0 ;
    for (int i := from ; i < to ; i++) { sum := sum + a[i] ; }
    return sum ;
}

void main() {
   Versioned⟨int⟩ a[1000] ;
   Versioned⟨int⟩ total := 0 ;
   for (int i := 0 ; i < 1000 ; i++) { a[i] := 1 ; } //initialize
   record { total := ParallelSum(a, 0, 1000) ; }
   assert(total = 1000) ;
   a[333] := a[333] + 1 ;
   repeat ;
   assert(total = 1001) ;
}
```

Figure 2. Simple example of applying self-adjusting computation to a parallel program using revisions.

we add 1 to element a[333] and do a repeat where the computed total is now 1001.

2.3 Execution

In Section 3, we present the algorithm to implement record and repeat in detail. This algorithm records the dynamic dependence graph of a parallel computation while it executes. When repeating a computation, the algorithm selectively reexecutes (and rerecords) only those revisions that have a dependency on changed input.

Revisions are the unit of memoization in our model. During the repeat step, revisions whose input dependencies have not changed are not reexecuted but rather *replayed* by consulting and applying their memoized effect.

The execution of our example program is shown in Figure 3. The left side shows the dynamic unfolding of the recursive computation of revisions during the record phase (in a so-called revision diagram). Note that memoization takes place at the level of the forked-joined revisions, not functions; revisions are reexecuted or replayed in their entirety.

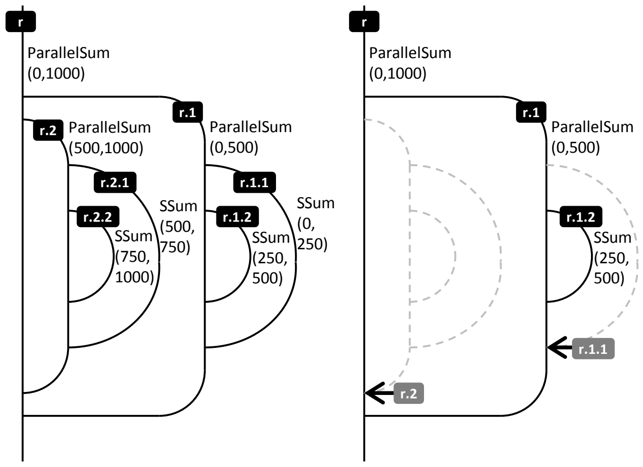

Figure 3. Revision diagrams illustrating the initial computation recorded by record (left) and the repeated computation performed by repeat. On repeat, the runtime avoids reexecuting some of the revisions, and simply replays their effects on join.

The right side of Fig. 3 shows the revisions that are reexecuted during the repeat phase (solid lines). This reexecution includes the revision that sums the range a[250−499] as well as that revision's ancestors. The dotted lines represent revisions whose behavior is not affected by the update to a[333]. Only two of these revisions inside the computation (r.1 and r.1.2) are actually reexecuted during the repeat phase.

During recording, we need to track memory accesses. Since the memory locations that are potentially involved in data dependencies have a special isolation type, we can avoid the excessive overhead that would result from tracking every single memory access.

2.4 Performance

We give a detailed evaluation of our algorithm and implementation in Section 4. In particular, we parallelized a multipass CSS layout algorithm using the concurrent revisions model and then applied the self-adjustment algorithm. We also studied two other interactive sample applications (a raytracer and a morph creator) and two smaller benchmarks (a webcrawler and a spell checker).

Just as it is sometimes difficult to write parallel programs that perform well, we found that effective use of self-adjustment requires some thought by the programmer. By studying the performance of our benchmarks, we identified specific programming techniques that enable self-adjustment to perform better in practice:

- The first important observation is that if the granularity of the smallest replayable unit is too small, the overhead of dependence tracking is too high. This issue is in perfect sync with the well-known importance of controlling the granularity of the smallest schedulable unit to limit the overhead of parallelization. Both of these issues can be addressed uniformly in our framework, by limiting the number of revisions forked (because in our model, revisions are both the smallest schedulable unit and the smallest replayable unit). In our example program, we control the granularity using a threshold parameter.

- The second observation is that if revisions access large amounts of data (for example, a bitmap representing a picture), tracking access to each individual memory location incurs too much overhead. To solve this problem, we introduce pseudovariables called *markers*. Each marker represents a group of locations, and we ensure that whenever revisions read/write a memory location represented by a marker, it also reads/writes the corresponding marker. Using such markers allows us to track dependencies more coarsely (trading precision for speed).

Using the granularity and marker techniques, we were able to achieve excellent performance on all sample applications:

- The recording overhead was very small and was in all cases more than compensated by the speed gained from parallel execution. On a eight-core machine, recording was thus still between 1.8x and 6.4x times *faster* than the sequential baseline computation.
- Repeating the computation after a small change was between 12x and 37x times faster than the sequential baseline computation (again on a 8-core machine).

2.5 Concurrent Revisions

This section briefly explains the concurrent revisions programming model. Adding self-adjusting computation to a concurrent imperative language with low-level synchronization primitives (such as threads and locks) is problematic. Racing side-effects introduce nondeterminism and cross-thread data dependencies that are difficult to track. To enable parallel self-adjusting computations, it is much better to start with a deterministic programming model that is not sensitive to a particular thread schedule.

Our work is based on a recently proposed deterministic concurrent programming model, *concurrent revisions* [14]. Preliminary results [14] indicate that (1) concurrent revisions are relatively easy to use for parallelizing reactive or interactive applications, (2) can deliver reasonable parallel performance for such applications, and (3) work well even for tasks that exhibit conflicts and would be difficult to parallelize by other means. The key design principles are:

- *Explicit Join.* The programmer forks and joins revisions, which can execute concurrently. All revisions must be joined explicitly.
- *Declarative Data Sharing.* The programmer uses special isolation types to declare data that may be shared.
- *Effect Isolation.* All changes made to shared data within a revision are only locally visible until that revision is joined.

Conceptually, the runtime copies all shared data when a new revision is forked, marks all locations that are modified in the revision, and writes the changed locations back to the joining revision at the time of the join. Therefore, the runtime can schedule concurrent revisions for parallel execution without creating data races. As a result, the computation is determinate, meaning that it is fully specified by the program and does not depend on the relative timing or scheduling of tasks [15].

To compare and contrast this model with classic asynchronous tasks, consider the example code in Fig. 4. In Fig. 4(a), we fork/join a classic asynchronous task; the assignment $\mathsf{x} := 1$ races with the assert statement above the join, with nondeterministic results. In Fig. 4(b), we fork/join a revision, and the variable x is declared to have the isolation type Versioned⟨int⟩. Since the writes to x become visible to the main revision at the join only, there is no race, and the the value of x is deterministic.

A technical report companion to the original paper contains a thorough discussion and formalization of the semantics of concurrent revisions [15, 16].

2.5.1 Revision Diagrams

It is often helpful to visualize computations using *revision diagrams* (see Fig. 4(c)). Such diagrams show each revision as a vertically aligned sequence of points, each point representing one (dynamic) instance of a statement. We use curved arrows to show precisely where new revisions branch out (on a fork) and where they merge in (on a join). In general, revisions can be nested and the life-time of a child revisions can exceed that of its parent. As such, it is more general than pure fork-join paralellism where the life-time of each task is limited to the lexical scope. Some restrictions still exist, where a revision diagram always forms a semi-lattice which is important when merging isolation types [16].

Since revisions can not directly communicate and are fully isolated, information flows along edges only. Revision diagrams overapproximate control and data dependencies: two statements not connected by a path in the revision diagram are always independent.

2.5.2 Cumulative Types

In initial applications for concurrent revisions [14], the versioned type Versioned⟨T⟩ delivered the correct semantics for the majority of data. However, this type is not appropriate for parallel aggregation, since it wipes out the current value of a location when joining a revision that modified that location.

For aggregation, the revisions model supports the *cumulative isolation type* Cumulative⟨T,f⟩, where f is a user-

(a)

```
int x;
x := 0;
Task t := fork{
  x := 1;
}
assert(x = 0
     ∨ x = 1);
join t;
assert(x = 1);
```

(b)

```
Versioned⟨int⟩ x;
x := 0;
Revision r :=
     fork{
  x := 1;
}
assert(x = 0);

join r;
assert(x = 1);
```

(c)

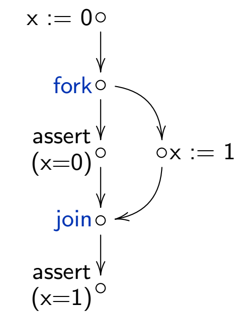

Figure 4. **(a)** A classic asynchronous task operating on a standard integer variable. **(b)** A revision operating on a versioned integer variable. **(c)** A revision diagram that visualizes how the state of x is branched and merged.

(a)

```
CumulativeInt x := 0;
r := fork { x := x + 2; }
x := x + 3;
join r; //x = 3 + (2 − 0)
assert(x = 5);
```

(b)

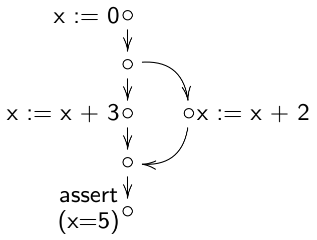

Figure 5. Example illustrating parallel aggregation with the isolation type CumulativeInt.

specified merge function f : (T,T,T) → T. This merge function is called by the runtime on join. For example, we can define a cumulative integer as

```
CumulativeInt = Cumulative⟨int,accumulate⟩
```

where the function accumulate is defined as follows:

```
int accumulate(int current, int joined, int original) {
    return current + (joined − original);
}
```

The runtime calls this function during a join with current being the current value in the joining revision, joined being the current value in the joined revision, and original being the value at the time the revision was originally forked. Fig. 5 shows an example of how CumulativeInt performs aggregation. Both Versioned⟨T⟩ and Cumulative⟨T,f⟩ are subclasses of a Versioned class.

Note that the determinacy of the computation does not depend on the merge function being associative or commutative; the only requirement is that it be a function in the mathematical sense.

3. Implementation

We now proceed to explain how we add support for self-adjusting computation to concurrent revisions. We provide high-level pseudo-code of our algorithm and the implementation of the main operations of read and write (of versioned locations), record, repeat, fork, and join.

Our presentation contains simplifying assumptions and restrictions that are not part of our actual implementation[1] since we believe that these details would make it quite challenging to follow the logic of the algorithm. We discuss some of the differences and the relationship to the concurrent revisions algorithm [14] in more detail in the appendix.

3.1 Motivation

We present detailed pseudocode since the algorithm proved difficult to get right and we went through multiple iterations. Moreover, it is significantly different from all previous implementations of self-adjusting computations, which were designed purely for sequential execution. Some particular difficulties include:

- The decision whether to reexecute or replay a revision must be both fast and precise. Simple validation schemes (e.g. recording all inputs and checking whether they are the same before replaying) are too slow, while simple invalidation schemes (e.g. mark a revision for reexecution any time anybody writes to one of its inputs) are too imprecise. Precision was particularly important for the CSS layout example.
- Dependencies must be tracked accurately. In a sequential setting, timestamps are sufficient to determine dependencies. In a concurrent setting, we need to track the structure of the computation (which is in our case represented by the revision diagram). Note that this approach crucially relies on the isolation guarantees.
- The data structures must be designed to be thread-safe (both record and repeat contain parallelism), yet use minimal synchronization to avoid overhead.

3.2 Description of Operation

At a high level, our implementation handles the example program in Fig. 2 as follows:

1. The call to record performs a computation as shown in the revision diagram in Fig. 3, on the left. As revisions execute, we construct a *summary tree*, where each summary records details about a revision, such as (1) a pointer to the entry point of the code, (2) the sequence of child summaries, in fork order, (2) the writes performed by the summary, and (3) the dependencies of the summary. Dependencies can be *external* (the summary depends on a

[1] Specifically, we assume that no revisions are used outside the record/repeat computations, we omit memory management considerations such as garbage removal and reference counting, we store writes directly in List data types (assumed to be thread-safe) rather than building truly thread-safe segment trees [14], we do not consider cumulative types as described in Section 2.5.2, and we do not discuss how we handle the creation of temporary or permanent versioned locations during recording.

value that was written before the recording began), or *internal* (the summary depends on a value that was written earlier during recording). When finished recording, the summary tree contains all the needed information. We assume throughout that all forked revisions are joined before recording ends.

2. Upon the assignment a[333] := 2 the runtime records that this location was written to.

3. When the user calls repeat, we first perform *external invalidations*. This means that for any summary that has an external dependency on a value that was modified since the last record, we invalidate that summary and all of its parents. In this example, the summary r.1.2 has an external dependency on the modified location a[333], so we invalidate summaries for r.1.2 and r.1 and r.

4. We now check if we can replay the computation. If the root summary r were still valid, we could replay the entire computation instantly (by writing 1000 to total). However, in this case, the top summary r is invalid, so we have to reexecute it. As we reexecute, we also construct a new summary.

5. When the re-executing code calls fork, we pull up the summary from the previous execution that has the same location in the summary tree as the summary that we are just about to fork. If a summary is still valid, we need not reexecute it and we can keep the old summary object. Otherwise, we reexecute and construct a new summary object. In this example, summaries r.1 and r.1.2 are reexecuted, but not r.1.1 and r.2.

6. When joining a summary that was not reexecuted, we reapply the effects as memoized during record at the join point. For r.1.1, the recorded effect is to add 250 to the sum. Similarly, revision r.2 is entirely elided and its effect (adding 500 to the sum) is reapplied at its join point. See Figure 16 in the appendix for how the MergeWrites function (from Figure 8) works in the presence of user-defined merge functions.

Perhaps the most intricate part of the algorithm is the propagation of *internal invalidations* (in the above example, there are no internal dependencies and thus no internal invalidations). We perform internal invalidations whenever we encounter a fork while reexecuting a summary; right before executing the fork (and checking whether its summary is still valid), we first examine what has been written by the current revision so far, and check if any downstream summaries need to be invalidated because of it.

A common case is where the reexecuting revision writes a different value than the previously recorded one. Another case is where it writes to a location that it did not write to before (and any part of the code that depended on an older write needs to be invalidated). Perhaps the most subtle case is where a location that was written to previously is not written to at all this time around (and any part of the code that depended on the previous write needs to be invalidated). To handle such cases, we explicitly pass along the previous summary tree while building a new one during re-execution.

Another subtlety is that we wish to only invalidate summaries that are forked *after* the invalidating write. To handle this, we need to index writes in a summary relative to the number of forks and use this index when tracking dependencies.

3.2.1 The Summary Class

Figure 6 shows the main data structure in our algorithm; a Summary. A summary captures the information we record about each revision.

Summaries store the entry point[2] of the recorded code in code (which may be needed for reexecution), and a thread thread that executes the code. The summaries form a tree structure that corresponds to the fork structure of the program. Each summary has a parent summary (null if it is the root of the tree), and a list of children summaries. The child summaries are ordered by the fork order in the summary's thread.

A summary s may be uniquely identified in the summary tree by a coordinate (d,i), where d represents the depth of the summary in the tree (with the root summary having depth 0), and i represents the index of s in the list s.parent.children (i is -1 if s is the root summary).

The execution of a code thunk by a thread is broken into *segments* by fork operations: a thread that forks n children has $n + 1$ segments. For each segment, the summary records in the writes field the last write operation to each versioned location written during the execution of that segment.

Each segment maintains an explicit write list[3]; we can then implement reads to versioned locations by searching through a parent chain. This gives the correct semantics for concurrent revisions where each revision only sees its own writes and is fully isolated from other revisions. Furthermore, we have a static variable globalwrites (also defined in Figure 6) that contains all the global writes (i.e. outside the recorded computation[4]). We assume that all versioned locations are present in the first map in the globalwrites list as if they were all initialized before the program starts.

The isValid flag tracks if the summary is still valid: if any dependency changes, this flag is set to false when dependencies are validated. The dependencies field maintains the dependencies of a summary explicitly. It is implemented as a map from versioned locations to coordinates in the summary tree. For example, if a location l was read in the summary's thread, there would be an entry for l with coordinates (d,i) if

[2] For simplicity, we assume that parameters are passed using versioned variables.

[3] Our actual implementation encodes these lists using thread-safe segment trees [14].

[4] Again, our actual implementation uses segment trees which enables us to use concurrent revisions outside the computation as well.

```
private class Summary
{
   // readonly fields
   Thunk code ;
   Summary parent ; // tree parent
   int depth, index ; // coordinate in tree

   // all fields below change during execution of code
   bool isValid ;
   Thread thread ;
   List⟨Summary⟩ children ; // tree children
   // read before write dependencies
   Map⟨Versioned, Pair⟨int,int⟩⟩ dependencies ;
   List⟨Map⟨Versioned,Value⟩⟩ writes ; // write segments

   int forkCount() {
      return children.Count ;
   }

   // constructor
   Summary(Summary parent, Thunk code) {
      this.parent := parent ;
      this.code := code ;
      this.depth := (parent = null) ? 0 : parent.depth+1 ;
      this.index := (parent = null) ? 0 : parent.forkCount() ;
      this.isValid := true ;
      this.children := new List⟨Summary⟩() ;
      this.dependencies
               := new Map⟨Versioned, Pair⟨int, int⟩⟩() ;
      this.writes := new List⟨Map⟨Versioned,Value⟩⟩() ;
   }

   private void Start(Summary prev) {
      if (thread ≠ null)
         error("cannot start a summary more than once") ;
      writes.Add(new Map⟨Versioned,Value⟩()) ;
      thread := StartNewThread{
         // ‘current’ and ‘previous’ are threadlocal state
         current := this ;
         previous := prev ;
         code() ;
      }
   }

   private void Wait() {
      if (thread ≠ null) thread.join() ;
   }
}

[threadlocal] private static Summary previous ;
[threadlocal] private static Summary current ;
private static List⟨Map⟨Versioned,Value⟩⟩ globalwrites ;
```

Figure 6. The definition of a Summary, including the (thread local) static variables current, previous, and globalwrites.

that location l was last written by an ancestor summary identified by (d,i); i.e. it was written by a prior summary at depth d in parent tree, just before its i^{th} fork point. Dependence on writes which occur at the top level, outside of the summary tree, are captured by the location $(0,-1)$.

Finally, we have the methods Start and Wait that start the thread associated with the summary and wait for it to end. The Start method sets two thread local variables, current and previous, before executing the associated code. The current thread local variable points to the summary itself while the previous thread local variable points to the summary constructed during a previous execution. On a first recording, the previous variable will be null.

The previous summary is used when a certain summary becomes invalid and is re-executed. By keeping the previous summary, we can compare the writes done in the new execution with the ones done previously. This information is necessary in order to precisely determine if more locations become invalidated when re-executing a summary. For example, if during re-execution a certain location is not written, but it was written during the previous execution, then all summaries that depended on that location need to be re-executed too.

3.2.2 Writing, Reading, and Dependencies

We assume that any writes and reads to versioned locations call the read and write operations defined in Figure 7.

A write operation on a versioned location updates the global store if its execution is outside of a computation being recorded or reexecuted (current=null). Otherwise, it writes to the last segment of the current (thread local) summary. In our real implementation, we implement such write lists using version maps in concurrent revisions, but for simplicity, we represent these lists here explicitly to model the concurrent revision semantics.

Similarly, a read operation on a versioned location reads from the global store if its execution is outside of a computation being recorded or reexecuted. Otherwise, the lookup operation proceeds up the tree of summaries, starting with the current summary. Note that when reading from each parent, we start looking at s.index in the writes list since we cannot see writes that occurred after our own fork point.

The other notable aspect of the read procedure is that it records write-to-read dependencies, when the write occurs outside of the summary in which the read took place using the AddDependency method. A dependency records the fact that a read of location l by summary s reads the value written by the last write to l within a particular *parent* segment in the summary tree; this segment is identified by its depth and index. The dependencies are stored in the dependencies map of the currently executing summary (see class Summary).

Now that we have described the summary data structure, the execution of reads and writes, and how they establish dependencies, we are in a position to describe the record, repeat, fork, and join procedures.

```
public void write(Versioned loc, Value val) {
  if (current = null) globalwrites.Last()[loc] := val ;
  else current.writes.Last()[loc] := val ;
}

public Value read(Versioned loc) {
    if (current = null)
        return LastWrite(globalwrites, loc) ;

    // was loc written to by this summary?
    if (FindWrite(current.writes, loc))
        return LastWrite(current.writes, loc) ;

    // otherwise, search along parent chain
    Summary s := current ;
    while (s.parent ≠ null) {
      for (i := s.index ; i ⩾ 0 ; i--)
        if (s.parent.writes[i].ContainsKey(loc)) {
            // internal dependency
            AddDependency(current, loc, s.parent.depth, i) ;
            return s.parent.writes[i][loc] ;
        }
      s := s.parent ;
    }

    // external dependency
    AddDependency(current, loc, −1, 0) ;
    return LastWrite(globalwrites, loc) ;
}
```

```
private bool FindWrite(List⟨Map⟨Versioned,Value⟩⟩ writelist,
                       Versioned loc) {
    for (i := writelist.Count−1 ; i ⩾ 0 ; i--)
        if (writelist[i].Contains(loc))
            return true ;
    return false ;
}

private Value LastWrite(List⟨Map⟨Versioned,Value⟩⟩ writelist,
                        Versioned loc) {
    for (i := writelist.Count−1 ; i ⩾ 0 ; i--)
        if (writelist[i].ContainsKey(loc))
            return writelist[i][loc] ;
    error("write not found") ;
}

private void AddDependency(Summary s, Versioned loc,
                           int depth, int index) {
    if (! s.dependencies.ContainsKey(loc)) {
        s.dependencies[loc] := (depth, index) ;
    }
}
```

Figure 7. Implementation for writing and reading versioned data, including the FindWrite, LastWrite, and AddDependency helper methods.

```
private static Summary compute ;

public void record(Thunk code) {
    compute := new Summary(null, code) ;
    compute.Start(null) ;
    compute.Wait() ;
    globalwrites.Add(new Map⟨Versioned,Value⟩) ;
    MergeWrites(compute.writes, globalwrites.Last())  ;
}

public void repeat() {
    DoExternalInvalidations() ;
    if (!compute.isValid) {
        var prev = compute ;
        compute := new Summary(null, compute.code)
        compute.Start(prev)
        compute.Wait() ;
    }
    globalwrites.Add(new Map⟨Versioned,Value⟩) ;
    MergeWrites(compute.writes, globalwrites.Last()) ;
}
```

```
private void MergeWrites(List⟨Map⟨Versioned,Value⟩⟩ joinee,
                         List⟨Map⟨Versioned,Value⟩⟩ main){
    foreach(segment in joinee) {
        foreach ((loc,value) in segment) {
                // writes in joinee win
                main.Last()[loc] := value ;
    }}
}

private void Invalidate(Summary s) {
    s.isValid := false ;
    // invalidate along the parent chain too
    if (s.parent ≠ null) Invalidate(s.parent) ;
}

private void DoExternalInvalidations() {
    var before := globalwrites.WithoutLastElement() ;
    foreach((loc,value) in globalwrites.Last()) {
      // skip if value has not changed
      if (value = LastWrite(before,loc))
        continue ;
      // invalidate dependent summaries
      foreach((s, (d, i)) where s.dependencies[loc] = (d, i)) {
        if (d = −1) // depth outside of summary tree
            Invalidate(s) ;
        }
    }
}
```

Figure 8. Implementation of record and repeat, and the helper methods MergeWrites, Invalidate, and DoExternalInvalidations.

3.2.3 Record and Repeat

The record procedure (Figure 8) takes a code thunk as input and creates a summary object that summarizes the execution of that thunk. After the execution of the thunk completes, the procedure merges the writes by the thunk into the global write list using MergeWrites. The MergeWrites procedure described in this section implements only the 'joinee wins' strategy for regular versioned locations. In the appendix we also show an implementation that can handle user defined merge functions (Figure 16).

The repeat procedure first performs invalidations via DoExternalInvalidations, which may set the isValid flag of various summaries to false. If the root summary is not valid, then a new summary must be created via (partial) re-execution, using the same code thunk as initially provided. The new root summary is started with the previous computed summary prev as its argument.

The procedure DoExternalInvalidations (see Figure 8) determines which summaries are invalidated by global writes that occurred between the record procedure and beginning of the repeat procedure. By construction, these writes occur in the last map of the list globalwrites. If the value of a location loc was written but has not changed then no invalidation takes place. Otherwise, for every summary s that is dependent on the location loc by a write with global scope (depth equal to -1), s is invalidated by a call to Invalidate.

The procedure Invalidate takes a summary as input and sets its isValid bit to false, and recursively invalidates the parent summary, until reaching the root of the summary tree.

3.2.4 Fork and Join

The fork operation defined in Figure 9 takes a code thunk as input. A fork operation may only take place in a thread that has a corresponding Summary (that is, no fork is allowed outside of a repeat/record execution). The fork operation first performs invalidations via DoInternalValidations. It then finds the (candidate) summary from the previous execution that corresponds to this fork. If this candidate is valid and has the same code thunk as input to the fork, then it can be safely reused/replayed. Otherwise, the forked code must be re-executed by creating a new summary. Finally, the summary corresponding to the fork is added to the current summary's children and a new write map is appended to accumulate updates in the next segment of the current thread's execution.

The procedure DoInternalValidations defined in Figure 10 determines which summaries must be invalidated due to differences in the behavior of the current thread segment (just before the fork) compared to its past behavior. The map recentwrites contains the locations written in the segment, while the map previouswrites contains the location written in the corresponding segment of the previous execution. For each location loc, there are three cases that (might) require invalidation:

1. loc is in the domain of recentw, but not previousw;
2. loc is in the domain of previousw, but not recentw;
3. loc is in the domain of both maps but has different values in each map;

```
public Summary fork(Thunk code) {
  if (current = null)
    error("no fork allowed outside computation");
  DoInternalInvalidations();
  // null if undefined
  Summary candidate := null;
  if (previous ≠ null ∧
        previous.children.Count > current.forkCount())
     candidate = previous.children[current.forkCount()];
  if (candidate ≠ null ∧ candidate.isValid ∧
       candidate.code = code) {
     candidate.parent := current;
  } else {
     var prev = candidate;
     candidate := new Summary(current, code);
     candidate.Start(prev);
  }
  current.children.Add(candidate);
  current.writes.Add(new Map⟨Versioned,Value⟩());
  return candidate;
}

public void join(Summary s) {
  if (current = null)
    error("no join allowed outside computation");
  s.Wait();
  MergeWrites(s.writes, current.writes.Last());
}
```

Figure 9. Implementation of fork and join.

When any of these cases occur, InvalidateDependentSummaries is called with that location as an argument. This procedure iterates over all summaries s that are dependent on loc. If s is already invalid, there is nothing to do Otherwise, we know that the summary s was previously or currently dependent because of a write to location loc in execution segment at coordinate (d,i).

The while loop searches up the parent chain in the summary tree to see if the summary current is "between" the coordinate (d,i) and the given summary s. More precisely, we mean that the (added, modified, or deleted) write to location loc at coordinate (current.depth,current.forkCount()) executes *at or after* the write to loc at coordinate (d,i) and *before* the read of location loc by summary s at coordinate (s.depth.s.index).

The variable dep is initially set to summary s. The while loop iterates as long as dep.depth is positive, greater than d, and greater than current.depth (if any of these conditions become false it is clear that the summary current cannot be between (d,i) and s).

```
private void DoInternalInvalidations() {
    if (previous = null ∨
        previous.writes.Count ⩽ current.forkCount())
      return ;

    var recentw := current.writes.Last() ;
    var previousw := previous.writes[current.forkCount()] ;

    // process recent writes
    foreach ((loc, value) in recentw)
        if (!previousw.ContainsKey(loc) ∨
            previousw[loc] ≠ value)
                InvalidateDependentSummaries(loc) ;
    // process the old writes that did not happen again
    foreach (Versioned loc in previousw.keys−recentw.keys)
        InvalidateDependentSummaries(loc) ;
}

private void InvalidateDependentSummaries(Versioned loc) {
    foreach ((s, (d, i)) where s.dependencies[loc] = (d, i)) {
        if (!s.isValid)
            continue ;
        Summary dep := s ;
        while (dep.depth > 0 ∧ dep.depth > d ∧
                dep.depth > current.depth)
        {
            if (dep.parent = previous) {
                if (current.forkCount() > dep.index)
                  break ;
                if (current.depth = d ∧ current.forkCount() < i)
                  break ;
                Invalidate(s) ;
                break ;
            }
            dep := dep.parent ;
        }
    }
}
```

Figure 10. Implementation of the helper methods DoInternalInvalidations and InvalidateDependentSummaries.

The loop body first checks if dep.parent is equal to previous. This check deserves some explanation. Certain summaries in the dependencies list of location loc may have been inserted during the current execution of repeat. Therefore, it is important to verify which tree we are in (the one from the past, before the execution of repeat, or the one currently being constructed) as we traverse up the parent chain. The check ensures that dep.parent is from the previous summary tree.

Now, if current.forkCount() is greater than dep.index then its execution order is after the summary s under consideration, and so it cannot occur between (d,i) and s. In this case, s will not be invalidated. Otherwise, the current.forkCount() is less than or equal to dep.index, and since we have that dep.depth > current.depth, we also know that the segment (current.depth, current.forkCount()) executes before s.

If current.depth is equal to d and current.forkCount() is less than i, the segment (current.depth, current.forkCount()) executes before the segment (d,i), which means that the write to loc in this segment, if repeated, would block the write from current reaching the read of loc in s. Otherwise, one of three cases hold:

- d is less than current.depth, in which case the summary current executes after segment (d,x) for any x.
- d equals current.depth and i is less than current.forkCount(), in which case the summary current also executes after (d,i);
- d equals current.depth and i equals current.forkCount(). In this case, we must assume that the write to location loc in segment (current.depth,current.forkCount()) comes after the write to loc in segment (d,i).

In all three of the case above, invalidation of summary s takes place.

The algorithm as presented is quite intricate and it would be useful to have correctness proof. We do not have a complete proof at this point, but we have worked through the most interesting parts of the reasoning with some care, and included it in the appendix (Section A.1).

4. Experiments and Results

In this section, we describe our experiences applying self-adjusting concurrent revisions to a selection of small applications. We give quantitative performance results and describe the programming techniques that were particularly effective at improving the performance of self-adjusting computation.

Our library prototype is an extension of the original concurrent revisions C# library, which uses an advanced work-stealing scheduler. We added primitives for record and repeat as described. To pass code arguments to the fork and repeat operations, we use *delegates*, the C# version of closures. Since our prototype is implemented entirely as a C# library, we were able to take existing sample programs and conveniently parallelize/incrementalize them using our new primitives.

4.1 Studied Examples

We studied two simple benchmarks (which we wrote ourselves), two interactive parallel sample applications (which we took from samples delivered with .NET 4.0 [30]), and a sequential CSS layout algorithm (which we obtained from a research group investigating efficient web browser designs). Each example contains a compute-mutate loop, for which we chose some representative small mutation.

Raytracer This interactive application (an extension of a sample from [30]) repeatedly renders a classic raytracer scene, showing polished balls on a tiled plane. The *computation* traces, for each pixel, an independent ray that touches various objects as it bounces around. The *small mutation* between repetitions is to alter the color of one of the balls, which changes the color of all pixels whose rays have touched that ball.

StringDistance This simple example is inspired by a spell-checker that provides correction suggestions. The *computation* finds for each of 20 given words the 3 closest matches in a dictionary of 12,000 words, using the Levenshtein distance metric (also known as edit distance) [31]. The *small mutation* is to add one word to the dictionary.

WebCrawler This example is inspired by a webcrawler, and (unlike all other examples) is I/O-bound rather than CPU-bound. The *computation* performs several tasks for each of 30 given URLs, such as measuring latency, downloading and validating the HTML, and counting the number of words. The *small mutation* is to modify one of the URLs. For this benchmark, we assume the content at the URLs is static.

CSS Layout This example models a CSS (cascading style sheet) layout algorithm, as employed by HTML browsers. While simpler than a real browser, it contains important complexities such as the presence of floating boxes. The original sample code we obtained was sequential; we parallelized it and added self-adjustment. The *computation* lays out a randomly generated tree containing about 400,000 boxes using three passes performed in sequence:

1. The first pass propagates down temporary width and height, and propagates up minimum and preferred width.
2. The second pass computes the width on the way down, and the height on the way up (laying out the children boxes). It also sets the relative position of all boxes.
3. The third pass computes the absolute position of all boxes.

The *small mutation* is to change the contents of one of the (leaf) text boxes, making it wider in the process. Depending on the data, this could in principle change the position of every single box, but in practice most boxes stay in the same position.

Morph This interactive application (an extension of a sample from [30]) computes a morph, that is, an animation that interpolates between two bitmaps guided by a list of vector pairs supplied by the user. The *small mutation* is to alter a small portion of each picture (in our case, by overlaying some text in each picture).

All examples with the exception of the layout algorithm exhibited plenty of parallelism and were easy to parallelize (by words, URLs, or picture tiles). In the layout example, divide-and-conquer naturally fits the tree structure, and can be applied in each of the three passes. The achievable speedup is modest, however, because the tree is not balanced.

In principle, one could spend extra development effort and manually derive incremental versions of all of these algorithms. However, we believe that for any realistically detailed application, the complexity of manual incrementalization can quickly become overwhelming. In the layout algorithm, in particular, understanding dependencies is challenging as they show up in many places (vertically for each pass, going down or up, and also laterally from one pass to the next).

4.2 Performance Results

We show the main performance results in Fig. 11. For each benchmark, the baseline column shows the time taken by the original code for the computation (without versioning, parallelism, or any form of self-adjusting computation). We then show execution times and speedups for recording the computation the first time around, and repeating it after making a small change to the inputs. All speedups are relative to the baseline, and all measurements are taken on a 8-core machine (a 2.33 GHz Intel Xeon(R) E5345). On the right, we show statistics about the computation: the number of summaries that were affected by the small mutation and needed to be reexecuted (out of the total number of summaries recorded), and the number of read (R) and write (W) accesses to versioned locations (V) and markers (M) that the computation performed during recording.

The numbers show that the recording overhead is small enough to be compensated by the gains from parallel execution. Specifically, recording is still faster than the sequential baseline, between 1.8x and 6.4x times. When repeating the computation after a small change, the speedup is much beyond the reach of optimal parallelization (8x), ranging from 12x to 37x. These numbers confirm our claim that self-adjusting computation is not just an alternative to parallelization, but indeed a lucrative extension.

To identify in more detail how the speedups are composed, we provide a more detailed breakdown and visual representation in Fig. 12. Again, we compare normalized execution times relative to the sequential baseline. For each benchmark we measure record and repeat, and also compute (which uses the vanilla concurrent revisions model, without recording dependencies). Moreover, for each of those scenarios, we test how much of the effect is due to parallel execution by imposing variable parallelism bounds on the underlying task scheduler: a parallelism bound of n means that at most n revisions can be simultaneously executing at any point of time.

We observe a few interesting details in Fig. 12. First, we can see that the use of the concurrent revisions model adds overhead: compute with parallelism bound 1 is up to

Benchmark	lines	baseline	parallel record		parallel repeat		reexecuted	RV	WV	RM	WM
Raytracer	1409	2.140 s	751 ms	**2.8x**	116 ms	**18x**	19 of 145	3.5M	0	0	0
StringDistance	618	268 ms	122 ms	**2.2x**	22.5 ms	**12x**	2 of 13	236k	236k	0	0
WebCrawler	525	16.3 s	3.42 s	**4.8x**	1.03 s	**16x**	3 of 109	135	135	0	0
CSS Layout	2674	185 ms	101 ms	**1.8x**	6.88 ms	**27x**	11 of 143	1674	377	2605	1296
Morph	2238	143 s	22.2 s	**6.4x**	3.86 s	**37x**	121 of 1729	0	0	5.7M	0

Figure 11. Main performance results and statistics. We show benchmark characteristics, average execution times and speedups relative to the sequential baseline, and statistics about the summaries and accesses to versioned locations and markers.

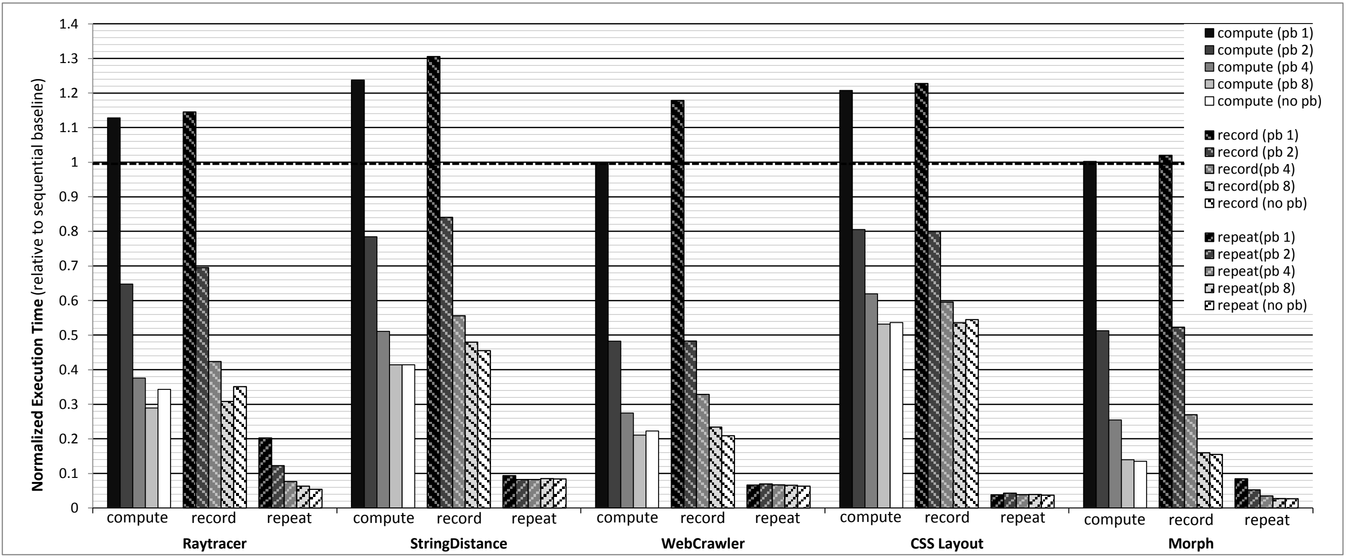

Figure 12. Graphical representation of normalized execution times for compute, record and repeat, relative to the sequential baseline. Lower is better. We show execution times for various parallelism bounds (for example, pb=4 instructs the revision scheduler to schedule at most 4 revisions in parallel).

Benchmark	Memory in VM		Working Set Size	
	self-adj.	baseline	self-adj.	baseline
Raytracer	7.55	2.59	91.7	90.3
Morph	2.62	1.13	127.2	131.6
Css0	169	126	368	281

Figure 13. Measured memory consumption for the three largest benchmarks, in Megabytes.

30% slower than the sequential baseline. However, the difference between compute and record is marginal, meaning that adding self-adjustment to concurrent revisions does not worsen this overhead significantly. We can also see that while parallelism is useful during repeat for two of the five examples (Raytracer, Morph) it does not help the other three, because they exhibit no significant parallelism within the parts of the computation that is reexecuted during repeat.

Memory Consumption. Running a self-adjusting computation does of course require more memory than the original baseline computation, since it records the computation and stores multiple versions of shared variables. To get a better

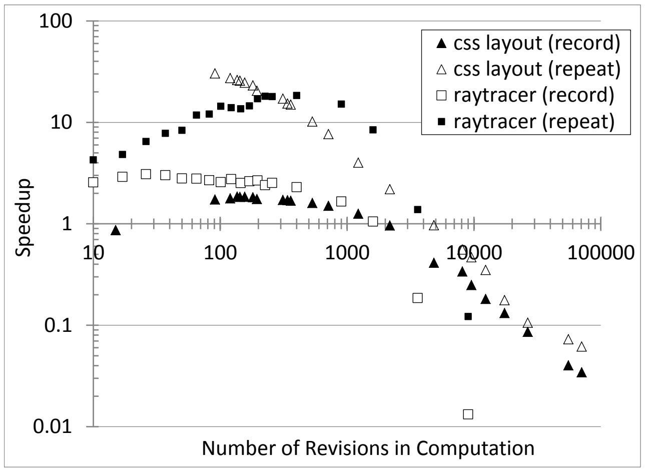

Figure 14. Effect of the granularity on speedups. The horizontal axis shows (on a logarithmic scale) the number of revisions in the computation (which increases exponentially when we decrease granularity). The vertical axis shows the speedup (or slowdown, if less than 1) relative to the baseline, also on a logarithmic scale.

understanding of the practical impact of this fact, we measured the following quantities for the three largest benchmarks (Fig. 13):

- *Size of the Recorded Computation*. We measure the memory allocated by the VM right after recording the computation, and compare it to the memory allocated by the baseline (which does not use versions or record computations) at the corresponding program point.
- *Comparison of Working Set Size*. We compare the working set size (as reported by the OS) of the self-adjusting computation and the baseline.

Not surprisingly, the size of the recorded computation varies with the benchmark. For the Raytracer and Morph benchmarks, it consumes a large proportion of the managed memory, but does not significantly affect the working set size. For the layout examples, the extra memory required is about 30%. Overall, we consider the extra memory consumption to be well within reason for many applications, considering the gains in speed.

4.3 Programming Techniques

Just like it is difficult to write parallel programs that perform well, we found that effective use of self-adjustment requires some thought on behalf of the programmer. By studying the performance of our benchmarks, we were able to identify specific programming techniques that enable self-adjustment to perform better in practice.

Revision Granularity. Since the creation and synchronization of tasks incurs a non-negligible overhead, parallel performance suffers if the task granularity is too fine. We addressed this issue by using parameters that control the number of revisions directly, or indirectly, e.g. by setting some threshold for recursive divide-and-conquer (such as shown in Fig. 3). Varying these parameters can affect execution times dramatically; we show this effect in Fig. 14. As can be seen, the speedups are best when using relatively few (less than 500) revisions only. This is true even for the repeat phase, where we may expect smaller granularities to pay off.

Simple Outputs. We found that for some locations, we can avoid versioning and improve performance (due to less indirection and copying). We call a location l a simple output if

- l is created before the compute/mutate loop.
- l is never read within compute(), and never written within mutate().
- l is race-free, i.e. not concurrently written to.

Simple outputs do not require versioning because (1) no computation in Compute depends on them, and (2) their value persists across iterations.

Markers. We found that we can improve performance considerably by tracking an entire group of locations as a whole, rather than each individual location separately. The idea is that

- The library provides a special marker class, with methods MarkRead() and MarkWritten().
- For each group of locations that the client code would like to track, it creates a single marker object.
- The client code ensures that whenever it reads from or writes to a location in the group, it calls MarkRead() or MarkWritten(), respectively.
- The client code ensures that all locations in the tracked group are race-free (i.e. they may not be concurrently accessed by two revisions, with at least one access being a write) and feedback-free (i.e. they may not be both an input to and an output of compute()).

If properly used, markers ensure sufficient invalidation of revisions to maintain the correct semantics of self-adjusting computation. We used markers in the layout algorithm, to track entire subtrees of the box trees as a single entity.

Clearly, it is desirable to reduce the reliance on such manual techniques as much as possible. We hope to automate more of this process in the future. For example, it is conceivable that lessons learnt in the context of parallelization, such as automated parameter tuning [8], can apply similarly for self-adjustment. Even so, we feel that the study of manual techniques is an important first step that can not be skipped.

5. Related Work

Our work spans the two research areas of (1) deterministic parallel programming, and (2) self-adjusting computation, and contributes to both. To the former, it adds the capability of incremental execution. To the latter, it adds parallelism, but also a shift in perspective that puts more control into the hand of the programmer and less burden on the compiler. It thus reflects a similar shift of perspective that has driven the parallel programming community towards parallel programming models and away from parallelizing compilers. Fig. 15 clarifies how to position our model with respect to previous work: the research presented in this paper spans the two left quadrants.

We now give some more detailed background on these research areas.

5.1 Incremental and Self-Adjusting Computation

An *incremental* algorithm efficiently computes its output with respect to the previous output and the changes in input. Compared to conventional (static) algorithms, an incremental algorithm has the potential for asymptotic improvements in speed for certain classes of changes to input. Traditionally, incremental computation has been explored in the context of functional languages. The primary techniques used to achieve incremental computation are dependence graphs,

Computation Structure	Declared by programmer	Inferred by compiler
Deterministic Parallel Programming	**Parallel Programming Models** - Cilk - DPJ - NESL - Concurrent Revisions	**Parallelizing Compilers** - Fortran - SUIF - Polaris - Paradigm
Self-Adjusting Computation	**Self-Adjusting Programming Models** - This paper - Early CEAL versions	**Self-Adjusting Compilers** - Delta ML

Figure 15. Categorization of Related Work.

memoization, and partial evaluation [5]. Dependence graphs track what computations must be updated upon changes to input [18, 26, 37]. Memoization (caching the result of function calls) [1, 6, 25, 29, 33] and partial evaluation (specializing a function with respect to some fixed input) [20, 36] can be used to achieve or enhance incremental computation. A comprehensive bibliography on incremental computation is provided by Ramalingam and Reps [34]. Incremental computation is increasingly important as the size of data sets increases and the computation over that data becomes more involved. For example, Guo *et al.* implemented memoization and dependence tracking in a Python interpreter to help scientists do faster prototyping of compute-intensive data processing scripts [23].

Our work builds on self-adjusting computation, a method for automatically obtaining an incremental algorithm from a batch algorithm [3]. The benefit of self-adjusting computation comes from not having to design an incremental algorithm, which is known to be quite tricky for even relatively simple problems. In the classical approach, data dependencies are tracked with *modifiable references*, and change propagation is efficiently implemented with *dynamic dependence graphs* and memoization. An early self-adjusting library, written in ML [2], was also reimplemented in a monadic style in Haskell [17]. In recent work, dependence tracking has been addressed at the data structure level, instead of individual memory cells, enabling further performance improvements [4].

5.2 Parallel Self-Adjusting Computation

Several recent papers have proposed parallel self-adjusting algorithms for specific problems [7, 11, 12, 35]. Unlike our work presented here, they do not investigate how to provide a general programming model. We are aware of only one paper addressing this same question [24].

In that work, the authors consider a tiny language that includes a **letpar** primitive for expressing the parallel evaluation of multiple expressions. They impose the restriction that locations be written at most once and are never read before written. While this restriction does indeed enforce determinism, it is more akin to dataflow models than to standard imperative parallel programming models (we discuss those in Section 5.3 below). The programming model we consider, concurrent revisions, is strictly more expressive, allowing multiple (and even concurrent) writes to shared locations.

Our work shares the idea from the Hammer et al. proposal [24] of tracking the sequential and parallel controlflow of a computation as a tree. In their work, they track dependences from writes to reads so as to later determine which reads are affected (read from a location whose contents have changed) by a change propagation algorithm. Our invalidation and re-execution approach over the tree of regions is similar to that of Hammer et al. However, our approach memoizes computations at the level of the concurrent revision (task), while the approach of Hammer et al. memoizes at the level of individual reads of locations. Thus, our approach can be much more coarse-grained and under the control of programmer (based on the programmer's choice of parallel decomposition using concurrent revisions).

The semantics of classical self-adjusting computation differs slightly from our loop characterization in cases where outputs of `compute` are overwritten by mutate, or where `compute` exhibits feedback (i.e. outputs are also inputs). For more detail, see Section A.2 in the appendix. We believe the loop characterization is more easily understood by programmers, and lends itself better to apply to existing compute-mutate loops.

5.3 Models for Deterministic Concurrency and Parallelism

Concurrent revisions [14] are a good fit for self-adjusting computation due to the determinism and well-defined semantics [15]. In recent years, a variety of related programming models for deterministic concurrency have been proposed that restrict tasks from producing conflicts. Some of these models leverage hardware support [9, 19], but do not guarantee isolation. Others support a restricted *fork-join* concurrency model [10, 13], or involve the implementation of efficient deterministic scheduling [32]. We believe that our approach to integrating self-adjusting computations with parallel programming could be realized in many of these frameworks. Another interesting connection between parallel execution and self-adjusting computation is that both appear to benefit from raising the abstraction level of data types [4, 27, 28].

6. Conclusion and Future Work

We have shown that a single, small set of primitives can enable programmers to write applications that can both (1) exploit parallelism, and (2) react incrementally to changes. We have presented the first known algorithm to perform such parallel self-adjusting computation. We have experimentally evaluated this idea by applying it to five example programs, and observe performance gains that are well beyond what can be achieved by parallelization alone.

Many questions remain to be answered by future work.

- Our current library implementation fully trusts the programmer to eliminate data races and invisible dependencies. More stringent checking may prove useful to find bugs, using a static approach as in DPJ [13] or a dynamic approach as in precise data race detection [21].
- Given the numerous programming techniques known for parallel programming, we believe the three we have presented for incremental programming barely scratch the surface. Knowing more about these techniques, and finding ways to apply them automatically, may help to further promote self-adjusting computation as an alternative to (or even better, an extension of) parallel computation.
- Similar techniques that we used to implement self-adjustment may also work to support speculative parallelism and an embedding of optimistic concurrency in the concurrent revisions model.
- We are considering user studies to evaluate our programming model and provide further evidence of its usefulness.

Acknowledgments

We thank Dan Grossman and Manuel Fähndrich for helpful comments and discussions.

References

[1] M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming (ICFP)*, 1996.

[2] U. Acar, G. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148:127–154, 2006.

[3] U. A. Acar. Self-adjusting computation (an overview). In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2009.

[4] U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Türkoğlu. Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*, 2010.

[5] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 32:3:1–3:53, November 2009.

[6] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Principles of Programming Languages (POPL)*, 2003.

[7] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Parallelism in dynamic well-spaced point sets. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2011. Symposium on Parallelism in Algorithms and Architectures.

[8] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Programming Language Design and Implementation (PLDI)*, 2009.

[9] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[10] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[11] P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[12] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: Mapreduce for incremental computations. In *ACM Symposium on Cloud Computing*, 2011.

[13] R. Bocchino, V. Adve, D. Dig., S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[14] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[15] S. Burckhardt and D. Leijen. Semantics of concurrent revisions (full version). Technical Report MSR-TR-2010-94, Microsoft, 2010.

[16] S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In *European Symposium on Programming (ESOP)*, 2011.

[17] M. Carlsson. Monads for incremental computing. In *International Conference on Functional Programming (ICFP)*, 2002.

[18] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Principles of Programming Languages (POPL)*, 1981.

[19] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared-memory multiprocessing. *Micro, IEEE*, 30(1):40 –49, 2010.

[20] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Conference on LISP and Functional Programming*, 1990.

[21] C. Flanagan and S. Freund. Efficient and precise dynamic race detection. In *Programming Language Design and Impl. (PLDI)*, 2009.

[22] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, 2009.

[23] P. J. Guo and D. Engler. Towards practical incremental recomputation for scientists: An implementation for the Python

language. In *Workshop on the Theory and Practice of Provenance (TAPP)*, 2010.

[24] M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2007.

[25] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Programming Language Design and Implementation (PLDI)*, 2000.

[26] R. Hoover. *Incremental graph evaluation*. PhD thesis, Cornell University, 1987.

[27] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Principles of Programming Languages (POPL)*, 2010.

[28] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *Programming Language Design and Implementation (PLDI)*, 2007.

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

[30] Microsoft. Parallel programming samples, .NET framework 4. `http://code.msdn.microsoft.com/ParExtSamples`, May 2010.

[31] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, 2001.

[32] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Notices*, 44:97–108, 2009.

[33] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages (POPL)*, 1989.

[34] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages (POPL)*, 1993.

[35] O. Sumer, U. A. Acar, A. Ihler, and R. Mettu. Fast parallel and adaptive updates for dual-decomposition solvers. In *Conference on Artificial Intelligence (AAAI)*, 2011.

[36] R. S. Sundaresh and P. Hudak. Incremental computation via partial evaluation. In *Principles of Programming Languages (POPL)*, 1991.

[37] D. M. Yellin and R. E. Strom. INC: a language for incremental computations. In *Transactions on Programming Languages and Systems (TOPLAS)*, volume 13, pages 211 – 236, 1991.

A. Appendix: Additional Material

In this appendix we include some additional material that discusses potentially interesting details that are however not crucial to the main contributions of the paper.

A.1 Correctness

The algorithm as presented is quite intricate and it would be useful to have correctness proof. Clearly, a full formalization of the algorithm and execution semantics is beyond the scope of this paper. However, we can give a precise correctness argument for our algorithm where we focus on the essential details. Our notion of correctness centers around the loop characterization described in the introduction. In particular, we leverage the key assumptions that the code executed by a summary is deterministic and that all threads are joined before the end of the outermost summary.

What we would like to show, is that after the initial call to record(code), calling record(code) subsequent times should have the same effect as repeat.

This correctness argument relies on two key lemmas. We say that a summary is "replayed" if we do not have to re-execute any of the code for that summary. Otherwise, it is "re-executed". Note that summaries may be replayed in either the repeat or fork methods. We say that two summaries are *deeply equal* if they have the same fields and the same tree structure. With these definitions, we can state the key lemmas:

LEMMA 1 (Record is neutral). *If we initialize isValid to false for every summary (so that every summary is re-executed), the behaviour is the same as if we just execute all the code associated with each summary. In other words, recording does not effect the computation.*

LEMMA 2 (Replay is sound).
If isValid for a summary is set to true at the join point for that summary, replay or re-execution produce summaries which are deeply equal. In other words, in Figure 9, after executing candidate.parent := current, then candidate is deeply equal to: candidate := new Summary(current, code); candidate.Start(prev);.

The first lemma is easy to validate by checking that each repeat and fork re-executes all code in each case and rebuilds each summary. The correctness argument for the second key lemma is more intricate though. We argue its correctness as follows:

- Assume there is an execution where the re-executed summary would produce a different value than the replayed one. So, there is an execution where the valid bit was set but re-execution got a different value.
- The only way the program can diverge from a previous execution is if a read returns a different value. So we need to show that if isValid is true, than all the reads are in fact returning the same values.
- Assume that there is a read which returns a different value. Consider the first such "divergent" read in in the partial order of segments enclosed by summaries. This partial order is with respect to forks and joins (a segment calling fork is before the fork, segment calling join is after the segment being joined). We perform a case analysis based on where the write initially came from:
  - If the write came from the same summary, then it could not have diverged by our determinism assump-

tion; all prior writes are deterministic. In the case where the value was written by a join, we leverage the fact that this is the first divergent read in the partial order. The summary being joined must not contain any divergent reads, and so, by induction, has deterministic behaviour.

- If the write came from a previous summary, the current summary must have been created with a call to fork. However, the call to DoInternalInvalidations inside of fork would have invalidated the current summary if either the write was from the same segment with a different value, from a different segment, or did not occur.
- If the write came from outside the summary tree, then it has either been replaced with a more recent write inside the summary tree or else a different value was written outside of the summary tree. In the first case, by the determinism assumption, the write must have come from a previous summary. However, DoInternalInvalidations would invalidate the current summary if a new write occurs. For the second case, the only way to reenter a summary tree after exiting is via the repeat method. In this case, DoExternalInvalidations would have set the isValid bit for the current summary if a different value had been written outside.

□

A.2 Compute-Mutate Loop Semantics

Our loop characterization can help to answer subtle semantic questions about self-adjusting computation in an imperative setting. It is not necessarily exactly equivalent to traditional notions of self-adjusting computation, in situations where either (1) outputs of the computation are overwritten by mutate, or (2) there is *feedback* (locations that are both inputs and outputs of the computation). Specifically, our loop characterization implies that in both of the following programs, the assertions do not fail:

```
int x = 0;                 int x = 0;
record { x := 1; }         record { x := x + 1; }
x = 2;                     repeat;
repeat;                    repeat;
assert(x = 1);             assert(x = 3);
```

Not all interpretations of self-adjusting computation in prior work behave in this way (the assertions may fail). There is often an implicit assumption that the programmer is responsible for identifying inputs and outputs, and must manually copy outputs back to inputs when feedback is desired.

A.3 Extending the Implementation for User-Defined Merges

In Section 3 we simplified discussion by eliding the case with user-defined merge functions. In this case, the merge function must be run when the summary is replaying or reexecuting. However, the MergeWrites function as presented does not have a way to calculate the three values needed to run a merge function. Figure 16 details an alternate version of MergeWrites which handles user defined merges.

In this case, MergeWrites takes as input the main summary which we are merging writes into and also the joinee summary we are merging writes from. Note that this version of MergeWrites is suitable for use in the join function; in the case of record and repeat, we extend this function to handle the lack of a current summary.

We need to work backward through the writes from the joined summary, only calling the merge function for the last write. To do this, we keep track of which locations we have already merged in the mergedLocations set. In order to find the original and current values for the location loc, we use the special methods GetValue and GetParentValue. GetValue cycles back through the summary parent chain to find the first location in which loc was written. This method is very similar to the read method, but does not add dependencies. The GetParentValue method starts immediately by looking for the value in the parent chain. It turns out that as a consequence of the concurrent revision model where revision diagrams always from a semi-lattice, the original ancestor value of the main and joinee is always the direct parent value of the joinee [16].

A.4 Pseudocode vs. Actual Implementation

We now discuss some of the differences between the presented pseudocode algorithm and our actual implementation in more detail.

Our pseudocode uses $\mathsf{List}\langle\mathsf{Map}\langle\mathsf{Versioned},\mathsf{Value}\rangle\rangle$ to store write sets. In the actual implementation, we use segment trees [14] (trees that have immutable nodes and mutable leaves) to encode the structure of the revision diagram. Each segment has a unique id. We then store inside each Versioned⟩ object a map from int to Value that stores the various versions. This map is encoded as an array of arrays of key-value pairs, for the sake of minimizing the synchronization requirements under concurrent access.

As described in [14], Segment objects (redundantly) maintain a list of versioned objects that contain versions for them; this allows us to remove those versions when deallocating segments. We deallocate segments when their reference count reaches zero.

To store the list of writes in a Summary, we keep a list $\mathsf{List}\langle\mathsf{Segment}\rangle$ of segments, with an extra first entry representing the root from which the revision was forked (if we have n segments, this list has thus size $n + 1$). We can then compute the write lists as needed by the algorithm by walk-

```
private void MergeWrites(Summary joinee, Summary main) {
    Set⟨Versioned⟩ mergedLocations = new Set⟨Versioned⟩();
    Map⟨Versioned, Value⟩ target = current.writes.Last();
    for (int i := joinee.writes.Count−1; i ⩾ 0; i--)
        foreach ((loc,value) in joinee.writes[i])
          if (!mergedLocations.Contains(loc)) // only merge last
                write
          {
            mergedLocations.Add(loc);
            Value mainValue := GetValue(main, loc);
            Value origValue := GetParentValue(joinee, loc);
            if (origValue = mainValue)
              main[loc] := value; // no conflict
            else
              main[loc] := loc.merge(mainValue, value,
                   origValue);
          }
}

private Value GetValue(Summary s, Versioned loc){
    if (FindIndex(s.writes, loc) ≠ −1)
        return LastWrite(s.writes, loc);
    else
        return GetParentValue(s,loc);
}

private Value GetParentValue(Summary s, Versioned loc) {
    while (s.parent ≠ null) {
      for (i := s.index; i ⩾ 0; i--)
        if (s.parent.writes[i].ContainsKey(loc)) {
            return s.parent.writes[i][loc];
        }
      s := s.parent;
    }
    return LastWrite(globalwrites, loc);
}
```

Figure 16. Merging writes with merge functions.

ing through the segment chain. Note that it is possible that several segments in this list are the same; this helps us to avoid excessive creation of empty write lists, which turns out to have a big performance impact (since the searching for writes tends to be expensive, and depends heavily on how many segments need to be traversed).

Finally, note that the foreach loop in DoExternalInvalidations, namely:

```
foreach((s, (d, i)) where s.dependencies[loc] = (d, i))
```

is potentially expensive since it visits all summaries in order find all dependencies. To keep the cost under control, we store the dependencies (which we presented as a field Map⟨Versioned,(int × int)⟩ inside summaries in the pseudocode) in the versioned object, as a Map⟨Summary,(int × int)⟩. This helps to iterate the loop above.

Similarly to the way we back-reference versioned objects that contain ids of Segment objects inside a list in the Segment objects, we redundantly store a list of Versioned objects in the summary objects, containing all the Versioned objects that contain that summary as a key in their dependency map. Again this helps to remove those dependency entries when a summary is deallocated.