JANUS: Exploiting Parallelism via Hindsight

Omer Tripp
Tel Aviv University
omertrip@post.tau.ac.il

Roman Manevich
The University of Texas at Austin
roman@ices.utexas.edu

John Field
Google
jfield@google.com

Mooly Sagiv
Tel Aviv University
msagiv@post.tau.ac.il

Abstract

This paper addresses the problem of reducing unnecessary conflicts in optimistic synchronization. Optimistic synchronization must ensure that any two concurrently executing transactions that commit are properly synchronized. Conflict detection is an approximate check for this condition. For efficiency, the traditional approach to conflict detection conservatively checks that the memory locations mutually accessed by two concurrent transactions are accessed only for reading.

We present JANUS, a parallelization system that performs conflict detection by considering sequences of operations and their composite effect on the system's state. This is done efficiently, such that the runtime overhead due to conflict detection is on a par with that of write-conflict-based detection. In certain common scenarios, this mode of refinement dramatically improves the precision of conflict detection, thereby reducing the number of false conflicts.

Our empirical evaluation of JANUS shows that this precision gain reduces the abort rate by an order of magnitude (22x on average), and achieves a speedup of up to 2.5x, on a suite of real-world benchmarks where no parallelism is exploited by the standard approach.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Experimentation, Measurement, Performance

Keywords speculative execution, concurrency, conflict detection, transactional memory

1. Introduction

Optimistic synchronization enables synchronizing parallel and distributed computations without the use of blocking. If such optimism causes improper synchronization, then the mis-synchronized work is undone and the entire system is restored to a consistent state.

Optimistic synchronization is a promising approach to parallelizing software. It offers ease of use (compared to writing explicitly concurrent programs) and clear correctness guarantees;

```
int work = 0;
/* parallel */ foreach (item in items)
   process(item, work);

process(Item item, int work) {
   work += weightOf(item);
   Result result = processItem(item);
   if (result.isSuccessful()) // item processed successfully?
      work -= weightOf(item);
   foreach (Item sitem in item.subitems)
      process(sitem, work); } // recursive call
```

Figure 1. Program with available parallelism that is hard to exploit

namely: atomicity, whereby the changes made by a transaction become visible to other transactions only at commit time; serializability, whereby committed transactions appear to execute in some serial order; and deadlock avoidance. Thanks to these strong guarantees, embodiments of this approach—in particular, software transactional memory [16] (STM)—have received much attention.

Optimistic synchronization is speculative in nature. Typically, when a conflict is detected between two concurrent transactions, one of the transactions is aborted, resulting in wasted work. Conflict detection is commonly carried out by checking whether the concurrent transactions access a mutual memory location, where at least one of them writes it. This is known as the *write-set* approach.

The write-set approach is efficient, but at the same time, overly conservative in ensuring serializability when the concurrent transactions—in their entirety—commute. Replacing the write-set approach for conflict detection, which tests every pair of operations performed by the concurrent transactions for commutativity, with a more refined detection algorithm, which considers a broader view of the concurrent histories in its commutativity judgments, can potentially reduce unnecessary aborts, thereby increasing the level of concurrency.

Illustrative Example Consider the program in Figure 1, inspired by the JFileSync application [2] (which we later discuss), where a collection of items is processed, and pending work (corresponding to items whose processing wasn't successful) is accumulated into variable `work`. Assuming that `processItem` is a pure function (i.e., free of side effects) and `processItem` calls are largely successful, the above loop admits a high degree of available parallelism. As long as updates to `work` are performed atomically, distinct items can be processed concurrently. Since most iterations restore `work` to its value at the beginning of the iteration, speculation is preferable to locking in ensuring the atomicity of accesses to `work`: If each

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'12, June 11–16, 2012, Beijing, China.

Copyright © 2012 ACM 978-1-4503-1205-9/12/06. . . $10.00

iteration is executed as a transaction, then in most cases, concurrent transactions are not in conflict having acted as the identity function on the shared state (`work`).

Using the write-set conflict-detection approach, however, conflicts would be detected between any pair of transactions whose executions interleave, causing (at least) one of the transactions to abort. In effect, the execution of all transactions will be serialized, resulting in slowdown (compared to sequential execution of the loop). In contrast, with more accurate detection, which considers *sequences* of memory accesses rather than single operations, the synchronization algorithm can exploit the observation that most transactions preserve the original value of `work`, which enables a high level of concurrency and a potential speedup (depending on the cost of internal bookkeeping and load balancing).

Current Approaches Recent techniques that we are aware of for improving speculative parallelization are challenged by this example. Transactional checkpointing [17], dependence-aware transactional memory [21] and elastic transactions [11] are all effective in reducing the cost and magnitude of aborts, yet these techniques cannot utilize the available parallelism in the above loop, which can only be appreciated by considering the (dynamic) interplay between increments and decrements of variable `work`.

Abstract locking [14, 19, 20] would also be futile in this case. Even if an abstraction specification is provided, transactions would conflict at the semantic level when attempting to increment and decrement variable `work` concurrently. Other forms of user specification, such as early release [15] or the annotations supported by the Alter system [24], are also of limited value here, because there is no particular conflict kind that can be universally suppressed without changing the semantics of the program.

Contributions Straightforward enforcement of sequence-based conflict detection is prohibitively expensive, especially relative to write-set-based detection. This paper describes an efficient technique for accomplishing such refined judgments while keeping runtime overhead on a par with that of the write-set-based approach.

The key to our technique is a *training phase* (Section 5.1), where sequences of operations that are likely to appear in conflict queries during parallel execution of the client application are extracted from single-threaded, synchronization-free, training runs of the application, and symbolic commutativity conditions are computed offline for pairs of such sequences.

For the information learned during training to generalize to new runs, our technique uses *generalization* (Section 5.2). This is done by mapping a concrete sequence (observed during training) to a regular abstraction of that sequence, such that commutativity conditions involving the original sequence remain valid for its abstraction.

For the information learned during training to be of practical value, our technique uses *projection* (Section 5.3), a commutativity testing algorithm ensuring that the dynamic context needed—during parallel execution—to check if a cached evaluation (learned during training) can be used for a commutativity query is essentially the same as in write-set detection: Only the read and write sets of operations are recorded. This also allows falling back to the write-set approach if the query cannot be answered from the cache.

We have implemented our sequence-based detection approach in JANUS, a parametric parallelization system based on optimistic synchronization (Section 4). The JANUS algorithm is abstract enough to encompass both memory-level statements and operations on abstract data types (ADTs), which enable further boosting of the detection algorithm's accuracy. In the latter case, the programmer specifies a representation function mapping a concrete data type to its abstract state, as well as the semantic interpretation of operations over the ADT.

Our evaluation of JANUS on a suite of five real-world benchmarks (Section 7) highlights the importance of sequence-based detection: JANUS achieved speedups of up to 2.5x, with few aborts, where its counterpart using write-set detection did not utilize any of the available parallelism in the benchmarks, and suffered from 22x more aborts.

The paper provides proof sketches for the correctness of the algorithms comprising our system. We defer full formal proofs to an accompanying technical report [22].

2. Motivation

The simple example of Figure 1 illustrates one of several semantic patterns appearing in real-world applications (cf. Table 5) that motivate refined, "sequence-based" reasoning about commutativity between histories. Below is a *non-exhaustive* list of such patterns, which we next illustrate using popular open-source applications:

Identity The transaction manipulates the shared state, but upon termination, restores it to its configuration prior to its execution.

Reduction A sequence of values is reduced to a single value by an associative and commutative operator.

Shared-as-local Shared memory is used as a "scratch pad". The transaction overwrites values set by other transactions, and only reads values written by its own execution. Programmers typically use this pattern as an optimization for reducing allocation overhead.

Equal-writes Distinct transactions assign the same value to a shared memory location. This pattern is useful when the condition enabling two (concurrent) transactions to write equal values can be checked efficiently at runtime.

Spurious-reads A read memory location cannot influence the transaction's observable effect (cf. [15]).

JFileSync The identity pattern is exemplified by Figure 2, where (a simplified version of) the JFileSync [2] file-synchronization utility is presented. The main loop of JFileSync iterates over pairs of files or directories recursively, and computes file synchronization metadata for each pair via the `compareFiles` call. Each iteration updates the number of items left to handle, as well as the overall "weight" of pending items, when a new work item is encountered (fields `itemsStarted` and `itemsWeight` of `monitor`, respectively). These values are popped once processing of that item completes.

Compile-time parallelization of this loop is impaired by data dependencies between iterations due to `monitor` and its fields. Moreover, even if the compiler is able to assert that `monitor` can be privatized, the shared `progress` object—responsible for interacting with "observers" of the computation (via calls to `fireUpdate`)—must remain shared for cancelation requests to apply to all active transactions. This mandates careful synchronization of concurrent accesses to `monitor`, which presents a challenge not only for the compiler, but also for developers attempting manual parallelization of this code. Speculative parallelization is thus appealing, but the write-set approach is too conservative.

JGraphT-1 Figure 3, showing the code of the JGraphT greedy graph-coloring algorithm, illustrates the spurious-reads pattern: Two transactions that execute in parallel starting from the same valuation of `maxColor` can only conflict if both assign new (and different) values to `maxColor`. If one (or both) of the transactions merely reads this variable, then there is no threat of conflict, and so read–write conflicts can safely be suppressed.

A compiler would fail to parallelize this algorithm because of loop-carried dependencies involving not only `maxColor`, but also the

```
/* parallel */ for (JFSDirectoryPair pair : pairs) {
  monitor.itemsStarted.add(2);
  monitor.itemsWeight.add(1);
  monitor.rootUriSrc = pair.getSrc();
  monitor.rootUriTgt = pair.getTgt();
  if (!progress.isCanceled()) { ...;
    weight = ...;
    monitor.itemsStarted.add(srcDirs.length+tgtDirs.length);
    monitor.itemsWeight.add(weight);
    progress.fireUpdate(); ...;
    /* compareFiles is recursive, also making "balanced" add-remove calls */
    compareFiles(...);
    monitor.itemsStarted.remove(monitor.itemsStarted.size()-1);
    monitor.itemsWeight.remove(monitor.itemsWeight.size()-1); }
  monitor.itemsStarted.remove(monitor.itemsStarted.size()-1);
  monitor.itemsWeight.remove(monitor.itemsWeight.size()-1);
  progress.fireUpdate(); }
```

Figure 2. JFileSync: *identity* (monitor.itemsStarted,itemsWeight) and *shared-as-local* (monitor.rootUriSrc,rootUriTgt)

```
int[] order = ...; // permutation of nodes fixing traversal order
int[] color = new int[neighbors.length];
int maxColor = 1;
BitSet usedColors = new BitSet(neighbors.length);
/* parallel */ for (int i = 0; i < neighbors.length; i++) {
  int v = order[i];
  usedColors.clear();
  for (int j = 0; j < neighbors[v].length; j++) {
    int nb = neighbors[v][j];
    if (color[nb] > 0) {
      usedColors.set(color[nb]); } }
  color[v] = 1;
  while (usedColors.get(color[v])) {
    color[v]++; }
  if (color[v] > maxColor) {
    maxColor = color[v]; } }
```

Figure 3. JGraphT-1: *shared-as-local* (usedColors) and *spurious-reads* (maxColor)

```
RuleSets rs = ruleSetFactory.createRuleSets(rulesets);
/* parallel */ for (DataSource dataSource : files) {
  String niceFileName = dataSource.getNiceFileName(...); ...;
  ctx.sourceCodeFilename = niceFileName;
  ctx.sourceCodeFile = new File(niceFileName); ...;
  rs.start(ctx); ...;
  rs.end(ctx); ...; }

/* in GenericClassCounterRule.java; called transitively by rs.start */
@Override
public void start(RuleContext ctx) {
  ctx.setAttribute(COUNTER_LABEL, new AtomicLong());
  super.start(ctx); }
```

Figure 4. PMD: *shared-as-local* (ctx)

```
Graphics2D g = ...;
/* parallel */ for (int index=0; index<nodes.size(); index++) {
  GraphNode n = (GraphNode) nodes.elementAt(index);
  if (n.nodeType == NORMAL) {
    g.setColor(this.getBackground().darker().darker());
    g.fillOval(x+n.x+...,y+n.y,nodeWidth, nodeHeight);
    g.setColor(Color.White); ...;
    g.drawString(n.lbl,x+n.x+...,y+n.y+nodeHeight/2+...); ...;
    g.setColor(Color.black);
  } else {
    g.drawLine(x+n.x+...,y+n.y,x+n.x+...,y+n.y+nodeHeight); }
  int x1, y1, x2, y2; ...;
  g.drawLine(x+x1, y+y1, x+x2, y+y2); ...; }
```

Figure 5. Weka: *equal-writes* (graphics)

shared color and usedColors arrays, where usedColors follows the shared-as-local pattern. These would also impair speculation based on the write-set approach. Manual parallelization of the code (e.g., using map reduction) is also not immediate, because the greedy algorithm mandates ordered traversal over the nodes of the graph.

PMD A near match for the shared-as-local pattern appears in PMD, a popular code scanner for Java whose main loop is shown in Figure 4. Most of the fields of the shared RuleContext instance, ctx, are treated as local by the loop's iterations. For example, each iteration first writes sourceCodeFile and sourceCodeFilename, and only later reads these fields. However, sharing between iterations is enabled via attributes stored in ctx (by calling {set,get}Attribute). This means that ctx cannot be privatized.

Compile-time parallelization of the PMD loop is thus inhibited. Manual transformation is also subtle: Calls to {set,get}Attribute occur deep in the call stack from the main loop, and are thus hard to track and observe. Sound handling of the persistent attributes further requires manual transformation of the RuleContext class, such that the fields that are amenable to privatization become separate from those that must remain global. Synchronization must then be added to mediate accesses to the global RuleContext state. Finally, write-set-based speculation will perform poorly because all iterations update the "local" sourceCodeFile and sourceCodeFilename fields.

Weka Last, Figure 5 presents code from the Weka data-mining library that is responsible for rendering a graph to a display device. This algorithm illustrates the equal-writes pattern in that distinct iterations accessing the same pixel do not conflict if they have set the Graphics object to the same color.

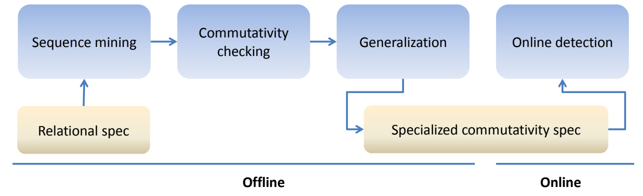

Figure 6. Outline of the JANUS architecture

Since the iterations are not invariantly independent, a compiler cannot statically parallelize this code, nor is it clear which manual transformations can be applied in this case. Speculative parallelization of the loop that is oblivious to pixel colors and only considers the accessed pixels is also overly pessimistic, but accounting for pixel values entails customization of the detection algorithm and may further lead to prohibitive instrumentation overhead.

3. Overview

The examples of Section 2 motivate more accurate conflict detection. However, boosting the accuracy of the detection algorithm comes at the cost of degrading performance. The write-set heuristic, employed in many realistic parallelization systems (e.g., [11, 14, 17, 20, 21]), tracks the read and write sets of a transaction, such that conflict detection reduces to a check whether there is a memory location that one of the concurrent transactions writes and the other accesses (either reading from it or writing to it).

Moving beyond the write-set approach, toward more accurate detection algorithms, involves two potential performance penalties: First, if the algorithm bases its judgment on information that is be-

yond read and write sets (e.g., the commutativity policies in [19]), then further instrumentation overhead—for collecting this additional information—is incurred. Second, the complexity of the detection algorithm itself may be prohibitive. This rules out naïve realization of sequence-based commutativity.

On the other hand, the codes in Figures 2–5—all real-world applications—illustrate several common scenarios where sequence-based reasoning provides significant value over the write-set approach. This remains true in the presence of state abstraction [14, 20], where conflicts are tracked atop the semantic state of data structures implementing ADTs. Though the (likely impractical) solution of instrumenting the program's state to record arbitrary trace information is theoretically possible, doing so does not only degrade performance, but also involves manual user intervention in deciding which information to record.

Our Approach In this paper, we present an automated technique for performing conflict detection over (potentially long) sequences, such that (i) there is no instrumentation overhead beyond that of the write-set approach, and (ii) the complexity of the detection algorithm is also comparable to write-set-based detection. Our technique, based on a preliminary learning phase, favors applications where learning holds promise, in that the effect of transactions on the shared state remains similar across different inputs. For other applications, where this is not the case, our approach will behave analogously to the write-set approach.

The flow of our solution is illustrated in Figure 6. The key aspect of our approach is offline learning of commutativity information: The application is profiled using training data, and commutativity conditions are evaluated for sequences arising during training that are expected to occur (frequently) in production mode. These candidate sequences essentially comprise dependent operations within the trace. During parallel execution, the conditions that were learned offline are used to test commutativity over sequences that are "similar" to the ones identified in training.

To ensure that the runtime overhead of our system is comparable to that of the write-set heuristic, the offline commutativity information we compute relates to sequences of operations over *individual* memory locations (Section 5.3). At runtime, this enables reconstruction of single-location-based sequences from the read and write sets of operations, and thus the dynamic context needed for sequence-based detection does not impose any instrumentation cost beyond that of the write-set approach.

The process prescribed by our approach consists of the following stages:

1. The input to our system is a user specification mapping concrete data structures to their (abstract) relational representation. The semantic state of a data structure is specified as a set of relations, and operations over the data structure are expressed using relational primitives. The `BitSet` class used in Figure 3, for instance, can be encoded as a 2-ary relation mapping integral values to boolean values. A relational description of the `get` operation is then specified as a `select` query, and similarly, setting the bit at index n to value x translates into removing the (unique) tuple whose first component is n and then inserting $\langle n, x\rangle$. The primary reason for the relational specification is that it facilitates reasoning about commutativity between sequences of operations.
2. Next, the subject application is exercised in "sequential" (i.e., single-threaded) mode using training inputs, such that no synchronization is required. In these runs, sequential dependencies are tracked between trace operations, and sequences of dependent operations that belong in different transactions are considered. In the example of Figure 3, for instance, two such sequences may be { `work+=2; work-=2; work+=1; work-=1;` } and { `work+=3; work-=3;` }.
3. Commutativity conditions—in the form of designated input states—are then computed for each pair of such sequences, where concrete values are substituted by symbolic values (e.g., { `work+=`x`;` `work-=`x`;` } for the second sequence). Generalization from concrete observations to arbitrary sequences is done using a theorem prover to ensure soundness.
4. To lift the learned commutativity information to new runs of the application, the sequences recorded during training are generalized into an abstract regular form by allowing arbitrarily many occurrences of idempotent subsequences within the generalized sequence (Section 5.2). The abstraction of { `work+=`x`;` `work-=`x`;` }, for example, is { `work+=`x`;` `work-=`x`;` }$^+$ (where $+$ denotes the Kleene-cross operator).
5. Finally, in production mode, a commutativity query is answered positively based on the cached information if the sequences in the query match a cached pair of sequences, and the input state belongs in the designated input states for the matched pair. If no match is found in the cache, then our technique falls back to the write-set approach.

4. The JANUS Parallelization Protocol

In this section, we define the protocol enforced by our parallelization system, where we leave the conflict-detection algorithm unspecified (until Section 5, which defines sequence-based conflict detection). We refer the reader to [22] for a formal description of the transition system underlying our protocol.

4.1 Parallelization Protocol

JANUS accepts as input (i) an initial configuration of the shared state, (ii) a list of tasks, each consisting of a program `prog` to be run and initial data values ($o \rightarrow \nu$), and (iii) a specification whether to commit the tasks in the order in which they were given. JANUS maintains two state variables: an atomic integer (*Clock*) implementing the versioning of the shared state, and a read-write lock ($lock$) for synchronizing concurrent activities. (In Figure 7, we denote use of the read (write) lock provided by $lock$ when invoking an operation via the "with $lock.r$ ($lock.w$)" syntax.)

JANUS repeatedly tries to execute the input tasks asynchronously, in parallel, until its task pool is drained (DOPARALLEL). Each execution attempt (RUNTASK call) encloses the argument task in a transaction, where the task's identifier and the time of creation of the transaction are recorded (CREATETRANSACTION). CREATETRANSACTION copies the global state (*Sh*), as well as the task-specific data ($o \rightarrow \nu$), into the local state of the freshly defined transaction. (Note that CREATETRANSACTION requires only a read lock, which enables multiple simultaneous transaction initializations.)

Next, RUNTASK executes the transaction sequentially (RUNSEQUENTIAL). In the case of in-order execution, the transaction can try to commit (after sequential execution) only once all its preceding transactions (ordered by task identifiers) have committed. As soon as this happens, or if the $ordered$ flag is not set, the transaction repeatedly attempts to commit until it either succeeds or fails. This is done by retrieving the sequence of operations committed between the time of start of the transaction (t.*Begin*) and the present time (now), and checking for conflicts between this portion of the system's history (ops^c) and the operations performed by the transaction (t.*Log*). (Note that no lock is acquired for conflict detection, which we later compensate for in COMMIT.)

If conflicts are detected between the histories, then the transaction aborts and RUNTASK will be called again from scratch. Otherwise, COMMIT is called. (Note that this is the only call that is governed by a write lock.) COMMIT first checks whether the considered system history has evolved between the point where DETECTCONFLICT was invoked and the present time (using the

```
The JANUS Parallelization Protocol

Types:
  Transaction = Record {
    tid                 : integer   // transaction identifier
    Begin               : integer   // transaction begin time
    Loc                 : state     // transaction-local state
    SharedPrivatized    : state     // privatized shared state
    SharedSnapshot      : state     // snapshot of shared state at Begin
    Log                 : list      // history of operations }
Inputs:
  ⟨⟨prog_1, o_1 → ν_1⟩, ..., ⟨prog_m, o_m → ν_m⟩⟩: tasks
  Sh_0: initial global (i.e., shared) state
  ordered: in-order execution flag

Variables:
  Clock: atomic integer initialized to 1
  lock: read-write lock with read lock r and write lock w

DOPARALLEL:
  for i := 1 ... m
    do asynchronously r := RUNTASK(prog_i, o_i → ν_i, i) while ¬r

RUNTASK(prog, o → ν, i):
  t := CREATETRANSACTION(o → ν, i) with lock.r
  RUNSEQUENTIAL(prog, t) // manipulates t.{Loc, SharedPrivatized, Log}
  if ordered wait until t.tid = GET(Clock)
  do while true
    now := GET(Clock) with lock.r
    ops^c := GETCOMMITTEDHISTORY(t.Begin, now)
    if DETECTCONFLICTS(t.SharedSnapshot, t.Log, ops^c)
      return false // abort
    if COMMIT(t, now) with lock.w
      return true

CREATETRANSACTION(o → ν, i):
  t := FRESHTRANSACTION (
    tid := i
    Begin := GET(Clock)
    Loc := (o → ν)
    SharedPrivatized := Sh
    SharedSnapshot := Sh
    Log := ϵ )

COMMIT(t, tcheck):
  now := GET(Clock)
  if now ≠ tcheck
      return false // abort
  INCREMENT(Clock)
  REPLAYLOGGEDOPERATIONS(t.Log, Sh)
  return true
```

Figure 7. Pseudo-code of the parallelization protocol enforced by JANUS, where the conflict-detection algorithm is unspecified

$now \neq tcheck$ test). If the history is valid (i.e., it hasn't evolved), then (i) $clock$ is incremented, and (ii) all the operations the transaction performed on privatized copies of global objects are replayed on their counterparts in global memory (REPLAYLOGGEDOPERATIONS). Otherwise, conflict detection is re-attempted.

Versioning To reduce the cost of state privatization, which is a key aspect of the JANUS algorithm, (fully) persistent data structures [10] can be used. A persistent data structure preserves the previous version of itself when modified; a data structure is *fully* persistent if every version can be both accessed and modified, which—in our case—permits concurrent modification of the shared state by multiple simultaneous transactions.

Conflict Detection The ideal test for conflicts is an explicit commutativity check. Assume that transaction t started running at time t_0, in state σ_0, and is performing a conflict test at time t_k. We denote by $\bar{b} = \langle b_1, \ldots, b_n \rangle$ the sequence of operations corresponding to transaction t. We refer to the operations corresponding to transactions that committed within time interval $[t_0, t_k]$, denoted by $\bar{a} = \langle a_1, \ldots, a_m \rangle$, as the *conflict history* of transaction t. Then t is in conflict with its conflict history iff

$$[\![\bar{a} \cdot \bar{b}]\!](\sigma_0) \neq [\![\bar{b} \cdot \bar{a}]\!](\sigma_0).$$

We say that a conflict-detection algorithm is *sound* if it does not permit a transaction that does not commute with its conflict history to commit. We say that a conflict-detection algorithm is *valid* if it does not prevent a transaction with an empty conflict history from committing.

Theorem 4.1. *Assume that the protocol of Figure 7 is instantiated with a sound and valid conflict-detection algorithm. Then*

1. *(**termination**) every run of the protocol is guaranteed to terminate if the sequential execution of all tasks (operation* RUNSEQUENTIAL*) terminates; and*
2. *(**serializability**) assuming termination, (i) every ordered run of the protocol is guaranteed to terminate in the same final state as its sequential counterpart, and (ii) every unordered run of the protocol is guaranteed to terminate in the same final state as a sequential execution of the tasks where their order of execution corresponds to the commit order in the concurrent run.*

Proof. For termination, observe that a RUNTASK call by task t can fail iff there is another task, t', whose RUNTASK call succeeds. Further observe that failure due to successful RUNTASK calls of other tasks can only occur finitely many times, since the number of tasks scheduled for execution is fixed and known in advance, which guarantees that a scheduled task will eventually commit.

For correctness, we rely on the soundness of the conflict-detection algorithm. In-order execution is enforced by preventing a transaction from attempting a commit until the global counter matches its identifier, which implies that all preceding tasks have committed already. □

5. Conflict Detection with Hindsight

The focus of this section is on the sequence-based conflict-detection algorithm employed by JANUS. The techniques enabling this algorithm are discussed in turn in the following subsections.

5.1 The Training Phase

The purpose of training is to specialize the detection algorithm in advance of parallel execution by building a cache of commutativity conditions relating to sequences of operations. In production mode, this cache saves the (expensive) work of performing sequence-based commutativity checking. Instead, the cache is searched for a match for the concurrent sequences in their input state. If no match is found, then the detection algorithm defaults to write-set-based detection.

Values and Dependencies. In support of dependence-analysis techniques that we introduce later in this paper, we make the assumption that the values assigned to objects are separable into subvalues. Formally, we assume a subvalue lattice with a partial ordering $v' \preceq v$, a join operation $\sqcup$, a meet operator $\sqcap$, and a subtraction operator defined for two ordered values $v' \preceq v$ by $v - v' \stackrel{\text{def}}{=} \min\{w \mid w \sqcup v' = v\}$.

Let *op* be an operation, such that the invocation of *op* in state σ yields state σ': $[\![op]\!](\sigma) = \sigma'$. Further, let $\langle o, v \rangle$ be an object-value pair in state σ, and $\langle o, v' \rangle$ its corresponding object-value pair in σ'. We denote the invocation of *op* in state σ by op_σ. The *frame* of op_σ's restriction to o, denoted as $op^f_\sigma(o)$, is the maximal unchanged subvalue of o: $v^f = v \sqcap v'$. The *written* subvalue of op_σ's restriction

to o is defined as $op^w_\sigma(o) = (v - v^f) \sqcup (v' - v^f)$. Finally, the *read* subvalue of op_σ's restriction to o, given by $op^r_\sigma(o)$, is the minimal subvalue $v^r \preceq v^f$, such that $[\![op]\!](\sigma) = [\![op]\!](\sigma[o \to (v \setminus v^f) \sqcup v^r])$. Intuitively, v^r is the portion of the frame that determines the outcome of op in state σ.

A dependency arises between two operation instances, op_{σ_1} and $\widetilde{op}_{\sigma_2}$, if there exists an object $o \in dom(\sigma_1) \cap dom(\sigma_2)$, such that

$$(op^w_{\sigma_1}(o) \sqcup op^r_{\sigma_1}(o)) \sqcap (\widetilde{op}^w_{\sigma_2}(o) \sqcup \widetilde{op}^r_{\sigma_2}(o)) \neq \bot. \quad (1)$$

That is, op_{σ_1} and $\widetilde{op}_{\sigma_2}$ both access a common subvalue of v, either for reading or for writing. (Input dependencies are subsumed by this definition.) We note that our definition of a dependency, inspired by [23], is general enough to apply both in a concrete semantics and in an abstract semantics induced by state coarsening.

Mining Sequences. We begin by explaining how the sequences are obtained. For a specific training payload, the training algorithm tracks dependencies between operations, within as well as across tasks, according to Equation 1. This results in a global dependence graph, $G = \langle V, E \rangle$, where the nodes of G are the operation instances from the run, and edge $v_1 \xrightarrow{l} v_2 \in E$ denotes that v_1 depends on v_2 over location l.

Next, for each location l accessed during the run, the (unique) maximal dependence path corresponding to l in G is retrieved. The path is then partitioned according to task boundaries. This yields a collection of subpaths, each corresponding to the operations involving l within a single task. These subpaths capture dependent sequences of operations that may participate, during parallel execution, in conflict queries.

We then compute commutativity conditions for each pair of such sequences. The conditions refer to the input state in which the sequences are evaluated. Our current prototype has limited symbolic capabilities, which we continuously evolve on demand. We presently account for equality and inequality tests appearing in the sequence, and further support certain useful distinctions that are particular to container ADTs (such as the presence of a key in a `Map` object).

5.2 Generalization via Sequence Abstraction

Concrete sequences of operations on shared locations often vary dynamically as a function of the input. For example, the add–subtract sequences induced by `work` in Figure 2 are length-wise proportional to the complexity of the input items. Caching commutativity information that holds for particular concrete sequences (those mined from sequential runs of the concurrent system) is thus of restricted value, as these sequences are tightly coupled to the training payloads.

To address this, we exploit the observation that distinct concrete sequences having the same "regular" description are, under certain conditions, conflict-wise equivalent. For instance, sequence { `work+=1;work-=1;` }—arising for the program in Figure 1— induces the same conflicts as { `work+=1;work-=1;work+=1;work-=1` }, and more generally, as any sequence in the language of regular expression ({ `work+=1;work-=1` })$^+$.

This is stated formally in the following claim, which relies on the notion of idempotence to assert a sufficient condition for conflict-wise equivalence between concrete sequences exhibiting the same regularity.

Lemma 5.1. *Let $s := s_1 \cdot s_2 \cdot s_3$ be a concrete sequence of operations, and assume that s_2 is an idempotent subsequence of s. Then operation* CONFLICT *from Figure 8 cannot distinguish between s and sequence $s' := s_1 \cdot s_2 \cdot s_2 \cdot s_3$. That is, given sequence $\overline{s}$, s conflicts with $\overline{s}$ in input state σ according to* CONFLICT *iff s' conflicts with $\overline{s}$ in this state.*

Sequence-based Conflict Detection Using Projection

Inputs:
 $\langle op^t_1, \ldots, op^t_m \rangle$: operations performed by current transaction
 $\langle op^c_1, \ldots, op^c_n \rangle$: committed operations
 G: entry state of current transaction

DETECTCONFLICTS:
 $m^t :=$ DECOMPOSE($\langle op^t_1, \ldots, op^t_m \rangle$)
 $m^c :=$ DECOMPOSE($\langle op^c_1, \ldots, op^c_n \rangle$)
 for $loc \in$ DOM(m^t) $\cap$ DOM(m^c)
  if CONFLICT($G, loc, m^t(loc), m^c(loc)$) return **true**
 return **false**

DECOMPOSE($\langle op_1, \ldots, op_k \rangle$):
 $subseqs := \emptyset$
 for $i := 1 \ldots k$
  $\{loc_1, \ldots, loc_l\} :=$ GETACCESSEDLOCATIONS(op_k)
  for $j := 1 \ldots l$
   $subseqs := subseqs[loc_j \to subseqs(loc_j) \cdot op_k]$
 return $subseqs$

CONFLICT($\sigma, l, \langle op_1, \ldots, op_k \rangle, \langle \overline{op}_1, \ldots, \overline{op}_w \rangle$):
 $r_1 :=$ GETREADSUBSEQUENCES($\langle op_1, \ldots, op_k \rangle$)
 $r_2 :=$ GETREADSUBSEQUENCES($\langle \overline{op}_1, \ldots, \overline{op}_w \rangle$)
 for $rsubseq \in r_1$
  if $\neg$SAMEREAD($\sigma, rsubseq, \langle \overline{op}_1, \ldots, \overline{op}_w \rangle$) return **true**
 for $rsubseq \in r_2$
  if $\neg$SAMEREAD($\sigma, rsubseq, \langle op_1, \ldots, op_k \rangle$) return **true**
 if $\neg$COMMUTE($\sigma, l, \langle op_1, \ldots, op_k \rangle, \langle \overline{op}_1, \ldots, \overline{op}_w \rangle$) return **true**
 return **false**

SAMEREAD($\sigma, l, \langle op_1, \ldots, op_k \rangle, \langle \overline{op}_1, \ldots, \overline{op}_w \rangle$):
 $\sigma'_1 := \langle op_1, \ldots, op_k \rangle (\sigma)$
 $\sigma'_2 := \langle \overline{op}_1, \ldots, \overline{op}_w \rangle \cdot \langle op_1, \ldots, op_k \rangle (\sigma)$
 $v_1 := \sigma'_1(l)$
 $v_2 := \sigma'_2(l)$
 return $v_1 = v_2$

GETREADSUBSEQUENCES($\langle op_1, \ldots, op_k \rangle$):
 $rsubseqs := \emptyset$
 for $i := 1 \ldots k$
  if ISREAD(op_i) $rsubseqs := rsubseqs \cup \{\langle op_1, \ldots, op_i \rangle\}$
 return $rsubseqs$

Figure 8. Algorithm for conflict detection using projection

Proof. Let σ' be the intermediate state after evaluating s_2 within s starting at entry state σ. Then if we evaluate s' at state σ, then the intermediate state *before* the second evaluation of s_2, as well as *after* it, is σ'. This implies that (i) the final states following evaluation of both s and s' at state σ are identical, and moreover, (ii) read operations within the first and second occurrences of s_2 within s' yield identical results. □

Generalization mitigates the problem of cache misses, and further allows use of "small" (yet sufficiently representative) inputs during training to accelerate cache population. Instead of relating commutativity conditions to concrete sequences, which mandates an exact match for a commutativity query arising at runtime, JANUS relates commutativity conditions to abstract sequences. These are computed offline, during the training phase, based on the concrete sequences due to the sequential runs. JANUS iteratively searches—in a bottom-up fashion—for idempotent subsequences within the concrete sequence, and applies the Kleene-cross abstraction to these subsequences.

5.3 Conflict Detection Using Projection

Figure 8 presents an "idealized" version of JANUS' projection-based algorithm for conflict-detection. The algorithm checks whether

the operations $\langle op_i^t \rangle_{i=1}^m$ of the current transaction, t, conflict with operations $\langle op_i^c \rangle_{i=1}^n$, which were committed by other transactions while t was running.

DETECTCONFLICTS first decomposes sequences $\langle op_i^t \rangle_{i=1}^m$ and $\langle op_i^c \rangle_{i=1}^n$ into multiple subsequences, such that every subsequence consists of the dependent operations accessing a single location. This is enabled by recording the read and write sets of each operations. Next, the sequences corresponding to shared locations (per the $loc \in \textsc{Dom}(m^t) \cap \textsc{Dom}(m^c)$ restriction) are analyzed for conflicts (in CONFLICT). Importantly, private locations—and the sequences of operations applied to them—are safely ignored, which for many realistic benchmarks is a dramatic saving.

A conflict is flagged between location-centric subsequences s_1 and s_2 induced by location l in one of two cases: Either (i) the value resulting from a read operation within s_1 (s_2) is influenced by whether s_2 (s_1) is performed before s_1 (s_2) or not (the SAMEREAD tests), or (ii) s_1 and s_2 do not commute over l.

The algorithm in Figure 8 is idealized in that in practice, CONFLICT consults its cache—populated during the training phase—instead of directly executing the SAMEREAD and COMMUTE checks. These are executed offline over the sequences due to training. If a miss is registered, then JANUS' default behavior is to fall back to write-set-based detection. Alternatively, JANUS can be configured to perform the sequence-based check online, which is unlikely to be acceptable in performance (though memoization can be used to support "online training").

Correctness While the location-wise commutativity check is natural, the SAMEREAD tests are less obvious. Lemma 5.2 affirms that together, these two checks pose as a sound transaction-wide commutativity judgment. The following example (in Java syntax) demonstrates that COMMUTE alone does not suffice:

```
x = 0, y = 0;
{ b = x==0; if (b) y = 1; x = 1; }    /* 1st transaction */
{ x = 1; }                             /* 2nd transaction */
```

In this program, the subsequences corresponding to both `x` and `y` commute. (`{ b = x==0; x = 1; }` commutes with `{ x = 1; }`, and `{ y = 1; }` trivially commutes with ϵ.) Yet the two transactions do not commute. This is because the (control) dependence between `x` and `y` is (incorrectly) ignored. The requirement that intermediate reads be unaffected by the order of execution of the concurrent transactions is a conservative approximation of the flow (through local state) between shared locations.

Lemma 5.2. *Let t_1 and t_2 be two transactions, such that t_1 commits during the execution of t_2. Further let $ops_1 = \langle op_1^i \rangle_{i=1}^m$ and $ops_2 = \langle op_2^i \rangle_{i=1}^n$ be the histories corresponding to t_1 and t_2, respectively, and σ the system state when t_2 begins execution. We denote by $\{ops_1^l : l \in L_1\}$ ($\{ops_2^l : l \in L_2\}$) the location-centric subsequences induced by ops_1 (ops_2), where L_1 (L_2) denotes the locations accessed by ops_1 (ops_2). Then t_1 and t_2 commute in σ if the following two conditions hold for every $l \in L_1 \cap L_2$:*

1. *(**commutativity**) ops_1^l and ops_2^l commute with respect to l: $([\![ops_1^l \cdot ops_2^l]\!](\sigma))(l) = ([\![ops_2^l \cdot ops_1^l]\!](\sigma))(l)$.*
2. *(**intermediate reads**) every read of l within ops_1^l (ops_2^l) results in the same value regardless of whether ops_1^l (ops_2^l) is evaluated in state σ or in state $[\![ops_2^l]\!](\sigma)$ ($[\![ops_1^l]\!](\sigma)$).*

Proof. We assume that the two conditions above hold, and prove for an arbitrary shared location l that its value is the same in both orders of execution of t_1 and t_2. First, we observe that control flow within t_1 (t_2) remains the same regardless of whether t_2 (t_1) is executed before t_1 (t_2). This is guaranteed by the invariance of intermediate reads from the shared state under the order of execution of t_1 and t_2, and these—assuming that the transactions are deterministic—are the only way of influencing the flow of a transaction. The projection of t_1's (t_2's) history on l therefore remains sequence ops_1^l (ops_2^l) regardless of the order of execution of t_1 and t_2. This, together with our assumption that ops_1^l and ops_2^l commute with respect to l in starting state σ, yields the conclusion that l's value is invariant under the order of execution of t_1 and t_2. □

The above claim, as well as its proof, extend naturally to the case where multiple transactions commit during the execution of a concurrent transaction.

Relaxed Consistency Beyond the baseline checks, and similarly to [24], JANUS supports a user-provided specification of consistency relaxations for data structures of choice. The user may specify that updates to (instances of) a data structure be committed in the presence of read-after-write (RAW) and/or write-after-write (WAW) conflicts involving the data structure's state.

The specification that RAW conflicts are tolerable for a data structure translates into dropping the SAMEREAD checks for shared locations defined by that data structure during conflict detection (cf. Figure 3). Analogously, tolerance of WAW conflicts is handled by dropping the final COMMUTE test (cf. Figure 4).

JANUS also performs limited automatic inference of relaxation specifications. For example, if out-of-order parallelization is permitted, then JANUS ignores WAW dependencies chaining two transactions, because these imply (under transitive reduction) that the transactions access the shared locations in conflict by first defining them and only then (potentially) reading them, and the final value of the shared locations is immaterial (as indicated by the out-of-order specification). This pattern is exemplified in Figure 4.

6. Relational Instantiation

In this section, we present a relational realization of the techniques described in Section 5. Representing object states and operations in relational form provides a natural way of computing the composite effect of a sequence of operations.

The relational instantiation is also general enough to encompass both standard (memory-level) transactions, whose executions consist of a sequence of statements, and transactions where certain data structures are equipped with abstraction specifications. These specifications enable treating method invocations atomically by considering their effect over abstract states.

The latter setting is similar in spirit to [20] and [14], where ADT semantics are taken into account in conflict detection to reduce the rate of spurious conflicts.

6.1 Preliminaries

Relations, Tuples and Dependencies We instantiate the definitions from 5.1 to the domain of relations. Assume a set v of untyped values drawn from a universe $\mathbb{V}$, which includes the integers: $\mathbb{Z} \subseteq \mathbb{V}$. A *tuple* $t = \langle c_1 : v_1, \dots, c_k : v_k \rangle$ maps a set of columns (c_i) to values from $\mathbb{V}$. We use $t\ c$ to denote the valuation of tuple t on column c. A *relation* is a set of tuples over identical columns. We define a partial ordering on relations $r' \preceq r \overset{\text{def}}{=} r' \subseteq r$ by the subset relation, join by set union, meet by set intersection, and subtraction by set subtraction.

We write $t \models \phi$ if tuple t satisfies formula ϕ, where ϕ is in the language of the grammar in Table 1. For any t, $t \models \mathbf{true}$, and similarly, $t \not\models \mathbf{false}$. Finally, $t \models c = v$ iff $t\ c = v$.

A relation r has a functional dependency (FD) $\delta := C_1 \to C_2$ if any pair of tuples in r that are equal on columns C_1 are also equal on columns C_2. We refer to C_1 (C_2) as the *domain* (*range*) of δ. We say that tuples t and t' *match* in relation r, and denote this by $t \sim_r t'$, if (i) r defines an FD c, such that t and t' are equal on all the columns in the domain of c, or (ii) r does not define any FD and t and t' are equal on all common columns.

ϕ	$:=$	$\mathbf{true} \mid \mathbf{false} \mid c = v \mid$	**(atomic)**
		$\neg\phi \mid$	**(negation)**
		$\phi \wedge \phi \mid \phi \vee \phi$	**(binary)**

Table 1. Production rules for formulas

Operation	Effect
`insert` r t	$r' = (r \setminus \{t' : t \sim_r t'\}) \cup \{t\}$
`remove` r t	$r' = r \setminus \{t\}$
$w :=$ `select` r ϕ	$r' = r, w = \{t \in r : t \models \phi\}$

Table 2. Primitive relational operations and their meaning

States and State Transformers In the relational representation, the value of an object comprises one or more relations: $o \mapsto \{r_i\}_{i=1}^{k}$. Each relation r_i is assumed to have at most one FD, where moreover, the domain and range of the FD partition the relation's columns. Intuitively, this specializes the relation as a function mapping "locations" to their associated "values".

The effect of primitive relational operations is listed in Table 2. `insert` first removes the tuples matching its argument tuple from its argument relation, and then adds that tuple to the relation. `remove` ensures that its argument tuple is not in the relation. Finally, `select` defines a relation containing the tuples from its argument relation that match the selection criterion, expressed as a propositional formula over the grammar of Table 1.

State transformers—both concrete and abstract—are expressed as sequences over the primitive relational operations. JANUS allows specifying different transformers for invocations of the same operation with different arguments.

6.2 Sequence-based Conflict Detection

We now explain how the techniques described in Section 5 are realized using the relational representation.

Tracking Dependencies The footprint of primitive relational operations is defined in Table 3. For sound tracking of dependencies between operations, we consider tuple t as belonging in the read set of operation `remove` r t if r does not contain t [23].

The footprint of a transformer, $\tau = \langle r_1, \ldots, r_k \rangle$, is the cumulative footprint of the relational operations it consists of:

$$\begin{aligned} write(\tau) = & \bigcup_{1 \le i \le k} write(r_i) \\ read(\tau) = & \bigcup_{1 \le i \le k} read(r_i) \end{aligned}$$

These definitions support the projection algorithm by enabling dependence-based decomposition of histories, as performed by the DECOMPOSE operation in Figure 8.

Equivalence Testing Both SAMEREAD and COMMUTE test for equivalence between different views of an object's value. COMMUTE considers the entire state of the object, whereas SAMEREAD considers only the relation defined by a read operation, which captures part of the object's state.

To perform equivalence judgments, we rely on a logical representation of the content of a relation, as specified in Table 4: The content of a relation is expressed as a restriction—formulated as a propositional formula—on the values from $\mathbb{V}$ contained in the relation. Thus, the removal of (the tuples in) relation w from relation r is reflected in the formula describing the content of r (ϕ^r) by conjoining it with the negation of the formula corresponding to w. Similarly, adding w to r is reflected as a disjunction from the formula describing the content of w.

The update rules corresponding to primitive operations are slightly more complex. For example, the rule for `insert` prescribes that first the tuples matching the tuple t to be inserted are removed. This is accomplished by conjoining formula $\bigvee_{c \in C_{dom}} c \neq t\ c$ with

Operation	Read Set	Write Set
`insert` r t	$\emptyset$	$\{t\} \cup \{t' \in r : t' \sim_r t\}$
`remove` r t	$t \in r\ ?\ \emptyset : \{t\}$	$t \in r\ ?\ \{t\} : \emptyset$
`select` r ϕ	$\{t \in r : t \models \phi\}$	$\emptyset$

Table 3. The footprint of primitive relational operations

Operation	Formula
$r' = r \setminus w$	$\phi^{r'} = \phi^r \wedge \neg\phi^w$
$r' = r \cup w$	$\phi^{r'} = \phi^r \vee \phi^w$
$r' = r \cap w$	$\phi^{r'} = \phi^r \wedge \phi^w$
`insert` r t	$\phi^{r'} = (\phi^r \wedge \bigvee_{c \in C_{dom}} c \neq t\ c) \vee \bigwedge_{c \in C} c = t\ c$
`remove` r t	$\phi^{r'} = \phi^r \wedge \bigvee_{c \in C} c \neq t\ c$
$w :=$ `select` r ψ	$\phi^w = \phi^r \wedge \psi$

Table 4. Logical representation of transformations on a relation

ϕ^r. Next, to express that t is then added to the relation, formula $\bigwedge_{c \in C} c = t\ c$ is disjoined from the intermediate formula.

Describing the content of a relation in propositional form allows implementing (symbolic) equivalence tests as calls to a SAT solver. Given two representations ϕ^r and ψ^r of the content of relation r, JANUS checks for equivalence between ϕ^r and ψ^r by asking the SAT solver for a satisfying assignment for $\neg(\phi^r \Leftrightarrow \psi^r)$. If the solver fails to find such an assignment (without timing out), then ϕ^r and ψ^r are confirmed to be equivalent.

7. Empirical Evaluation

In this section, we describe our prototype implementation of JANUS, as well as its experimental evaluation on a suite of five real-world benchmarks, where we tested (i) the gain from sequence-based conflict detection (compared to standard detection), and (ii) the significance of basing this mode of detection on offline training (with and without sequence abstraction).

7.1 Preliminaries

Prototype Implementation JANUS is implemented as a (static) Java library that exposes an interface for running client-provided tasks in parallel (via the `run`, `runInOrder` and `runOutOfOrder` methods), as well as for controlling various aspects of the execution (e.g., enabling profiling, configuring the profiling policy, setting the number of threads, modifying the thread management policy, etc).

For logging purposes, JANUS automatically inserts instrumentation hooks into its client application at runtime. For this, we reused functionality from Chord [1], a framework for transforming and analyzing Java bytecode based on the Joeq compiler infrastructure [25] and the Javassist bytecode-instrumentation library [9]. JANUS also uses the Sat4j SAT solver [5] for resolving equivalence queries (cf. Section 6.2).

The current JANUS prototype does not automatically identify candidate loops for parallelization. Instead, the user decides where to apply JANUSby manually transforming a loop structure into a JANUScall. (This decision can be guided by automated tools, such as [23], which we are planning to integrate into JANUS.)

JANUS accepts user-provided abstraction specifications as extensions of its core algorithm written in Java. When an instance o of a specified data structure is created, JANUS maps it to a fresh set of relations (per the specification). Later, when an operation is invoked on o, JANUS applies the model associated with the operation to the (abstract) relational state of the object, thereby accounting for the effect of the operation. The binding between concrete execution events and the user specification is handled automatically (via instrumentation).

Benchmarks We used five real-world benchmarks, all taken from the SourceForge code repository [6], for our experiments. These are listed in Table 5. All the benchmarks are sequential (though PMD has an execution mode where certain subcomputations are run in parallel), and their core functionality involves processing a collection of items in a loop. (This obviated the need to select which loop to parallelize.)

In JGraphT, we considered two distinct algorithms within the `GreedyColoring` entry point: (i) an optimized version of the greedy algorithm for graph coloring (`color`), and (ii) an implementation of the saturation-degree heuristic for node ordering for the greedy coloring algorithm (`largestSaturationFirstOrder`). In JFileSync, we used the `JFSComparison` entry point, which computes the differences (at all hierarchies) between pairs of directories. In PMD, we considered the main entry point (`PMD`), which iterates over a collection of Java source files and analyzes them (intraprocedurally) for correctness and quality problems. Finally, in Weka we used the `GraphVisualizer` entry point, which renders a graph to a display device by traversing its nodes.

For each of the benchmarks, we manually transformed the relevant loop by casting its iterations into task objects and replacing the loop construct by a call to JANUS to run the tasks in parallel. These transformations proved straightforward, being purely syntactic, and can be automated with relative ease.

Experimental Setup Our experimental design consists of two sets of experiments. In the first, we compared between two versions of JANUS whose sole difference lies in whether or not sequence-wide reasoning is used in terms of overall performance and retry rate. The *write-set-based* version checks for conflicts using the standard approach of breaking the concurrent histories into their constituent operations and testing each of the resulting operation pairs for conflicts. The *sequence-based* version instead embodies the detection algorithm of Section 5. We stress that the *write-set-based* algorithm is implemented as a subset of its *sequence-based* counterpart, which cancels out differences due to implementation choices (e.g., code optimizations).

The second experiment is concerned with the significance of offline training, and measures the fraction of misses in sequence-based conflict queries as an indication of how well JANUS generalizes from training runs. For pair $\langle s_1, s_2 \rangle$ of sequences, a miss occurs (during a production run) if there is no matching entry in the commutativity specification built during offline training. In our measurements, we count only unique queries (and thus multiple hits/misses for the same query are counted as one). Our evaluation comprises two modes of detection: with and without sequence abstraction (Section 5.2), which provides insight into the value of this technique.

In our experiments, we ran JANUS 5 times in training mode and 10 times in production mode. All the measurements prsented in this section are based on averages over all production runs excluding the first (cold) run. Inputs for the training and production runs were based on available clients of the benchmarks (mainly unit tests). A characterization of the inputs appears in Table 6. For scalability, we authored abstraction specifications for shared data structures. To identify these data structures, we used the Hawkeye tool [23].

We conducted our experiments using an IBM J9 V1.6.0 VM running on a Linux machine with an Intel Core i7 920 (Nehalem) processor that has four 2.67GHz cores, each multiplexing 2 hardware threads, for a total of 8 threads.

7.2 Performance Results

Figure 9 presents the speedup results for each of the benchmarks using 1–8 threads with both versions of JANUS. The ratio of overall retries to the number of transactions is shown in Figure 10. Results on cache misses in the 8-threads configuration appear in Figure 11.

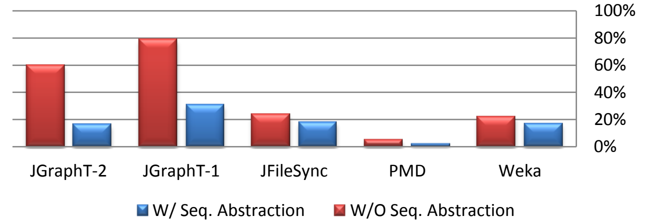

Figure 11. Misses: the ratio of misses to queries with and without the sequence-abstraction technique of Section 5.2

Speedup and Retries The performance results shown in Figure 9 demonstrate the dramatic effect of sequence-based reasoning: The sequence-based version of JANUS achieves an average speedup of 1.5x on 8 threads, where JFileSync enjoys a speedup of close to 2.5x. Conversely, the write-set-based version degrades performance with an average slowdown of 1.66x (or, equivalently, speedup of 0.6x) on 8 threads. These trends hold not only on average, but also for each of the benchmarks individually.

JGraphT-2 is the only benchmark where speedup under sequence-based detection is negligible. Our analysis indicates that this is because the transactions in this benchmark make intensive access to shared memory (comprising 6 data containers) all across their execution. Sequence-based detection is highly effective for this benchmark, as Figure 10 clearly indicates, but the JANUS protocol is geared toward transactions that make infrequent access to shared memory, such that the costs of privatization and fusion are amortized by the local work performed by the transaction.

The speedup data is compatible with the retry statistics presented in Figure 10. For all benchmarks, the number of retries under write-set-based detection is prohibitive: For PMD and JGraphT-2, the number of retries is directly proportional to the number of tasks, regardless of the number of threads. For the remaining benchmarks, the number of retries increases with the number of threads, the most extreme case being JGraphT-1 where the average number of retries per task is 4 with 8 threads.

In contrast, sequence-based detection yields a very low retries-to-transactions ratio, the average being 0.07 compared to 1.51 with write-set-based detection, which is a 22x improvement. A partial explanation for this significant gap between the detection algorithms is the commutative patterns exhibited by the benchmarks, which are listed in Table 5.

Misses The data on misses in Figure 11 indicates that (i) the commutativity specification due to the training runs generalizes well to production runs under sequence abstraction, but (ii) without sequence abstraction, generalization deteriorates significantly. More concretely, the average miss rate is less than 17% with sequence abstraction, with no more than 30% misses for all benchmarks (the worst being JGraphT-1), and 38% without applying abstraction to the sequences observed in training, where JGraphT-1 exhibits approximately 80% misses in this configuration. Sequence abstraction enables a 2.24x improvement in generalization from training runs, which points to the significance of this heuristic.

Sequence abstraction is most useful for the JGraphT benchamrks. For the other benchmarks, generalization is already robust even without this heuristic ($\leq$24% misses), which leaves little room for improvment. Indeed, access patterns to shared memory are highly dynamic in the JGraphT algorithms (e.g., the calls to `usedColors.set` inside an inner loop of variable length in Figure 3), making sequence abstraction significant for these benchmarks.

Discussion Manual or static identification of commutative patterns in the benchmarks we've considered can be challenging, if

Name	Version	Description	Prevalent Patterns
JFileSync [2]	2.2	Utility for synchronizing pairs of directories	Identity
JGraphT [3]	0.8.1	(1) Greedy graph-coloring algorithm (2) Saturation-degree node-ordering algorithm for heuristic graph coloring	Shared-as-local, spurious-reads Shared-as-local, equal-writes
PMD [4]	4.2	Java source code analyzer	Shared-as-local, reduction
Weka [7]	3.6.4	Machine-learning library for data-mining tasks	Equal-writes

Table 5. Benchmark characteristics

Benchmark	Input Description	Training Data	Production Data
JFileSync [2]	List of directory pairs	Random lists of length 5	Random lists of length 25
JGraphT [3]	Parameters for creation of random simple graph	100 nodes; average degree of 5	1000 nodes; average degree of 5
PMD [4]	List of Java source files	Random lists of length 10	Random lists of length 100
Weka [7]	Parameters for creation of random Bayesian network	100 nodes; average degree of 10	1000 nodes; average degree of 10

Table 6. Inputs for training and production runs

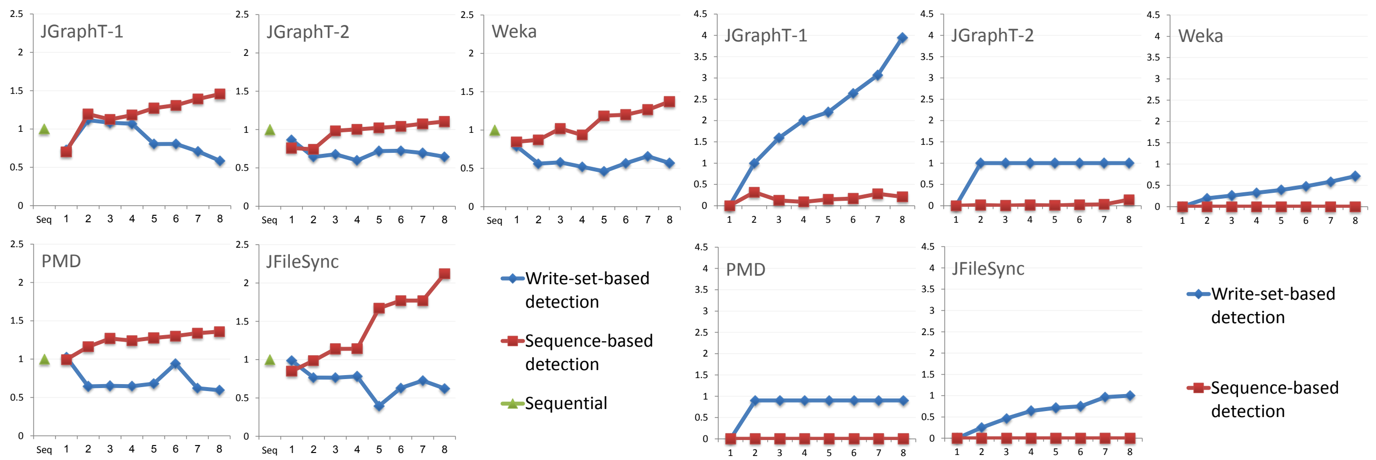

Figure 9. Speedup: the ratio of sequential to parallel execution times as a function of the number of threads (1–8)

Figure 10. Retries: the ratio of overall retries to the number of scheduled tasks as a function of the number of threads (1–8)

not impossible: First, these benchmarks are all of real-world scale (well beyond 100K LOC). They comprise multiple abstraction layers, and the behaviors that induce commutative patterns typically span nontrivial interprocedural control flows (cf. Figure 4).

Moreover, in some cases the patterns do not hold invariantly, or hold only for some of the conflicted memory locations. This is the case, e.g., with JGraphT-2, where the ordering algorithm manipulates several shared arrays whose access patterns are determined dynamically, according to the input graph. JANUS exploits the available parallelism, which cannot be identified statically or captured concisely via a pattern, but can be effectively leveraged based on data from training runs (only 16% misses for JGraphT-2).

Though the speedup for JGraphT-2 in specific is modest, we are encouraged by our overall performance results. While the speedups we've obtained are not directly proportional to the number of threads, we have managed to considerably improve the performance of four out of the five *real-world* benchmarks we experimented with (by 1.6x or more) via automatic parallelization.

We also believe that these results can be further improved: First, our prototype implementation of JANUS can benefit from more careful engineering. For example, our current implementation doesn't reclaim the logs of garbage transactions whose concurrent transactions have also terminated, and implements privatization transformations in a naïve fashion, instead of using full persistency. Second, the hardware we used does not enable full 8-thread parallelization: The 8 available hardware threads are supported by only 4 cores, each multiplexing 2 threads.

8. Related Work

There is an extensive body of research on synchronization in general, and optimistic synchronization in specific, that dates back to Bernstein's work from the 1960s on the use of commutativity in concurrency control [8]. Due to space constraints, we restrict our discussion to closely related research.

Use of Abstraction Herlihy and Koskinen [14] develop *transactional boosting*, a methodology for transforming linearizable objects into transactional objects. Boosting leverages ADT semantics for more accurate conflict detection by avoiding spurious conflicts due to the ADT's concrete realization. The *Galois* parallelization system [20], designed to exploit available parallelism in irregular applications, embodies a similar approach.

In [18], Koskinen et al. define a unified framework for describing both memory-level transactions and transactional boosting. Their framework allows conflict detection both at the concrete level, where simple statements are tested for commutativity, and at the abstract level, where the abstract effect of ADT operations is considered. Similarly, Tripp et al. [23] describe a unified framework for simultaneously tracking both concrete and abstract dependencies in sequential executions of a program, where concrete dependencies involve simple statements and abstract dependencies are due to a semantic conflict between ADT operations. The goal of this analysis is to uncover dataflow impediments to parallelization, where abstraction is used to reduce the rate of false impediments.

JANUS also enables state abstraction being parameterized in the state representation: It can track conflicts atop concrete program states, but the user can also apply abstraction specifications for shared data structures. A unique feature of JANUS, beyond state abstraction, is sequence-wide reasoning: JANUS is able to account for complete histories of operations during conflict detection.

Sequence-based detection is expressible in the (general) *gatekeeping* [19] conflict-detection paradigm, where the gatekeeper keeps track of all the actions performed by a transaction to enable state rollback, as well as to evaluate commutativity conditions that refer to the history of the computation. JANUS can be seen as the first (automated) realization of the gatekeeping scheme, which is—in general—nontrivial to implement, as the user is required to write explicitly concurrent (and highly optimized) synchronization code.

Concrete STM Various techniques have been proposed for avoiding false aborts and/or reducing their cost in standard STM. We mention some of these techniques below.

Dependence-aware transactional management [21] delays the decision whether to abort a conflicted transaction until any of the concurrent transactions in conflict with it commits. This is done by maintaining the dependency relation between transactions, such that if the conflicted transaction depends only on transactions that have aborted, then it need not abort.

Transaction checkpointing [17] reduces the cost of aborts by taking snapshots of intermediate (local and global) states during the execution of a transaction. Similarly, an *elastic* transaction [11] is able to respond to a conflict situation by dropping its work thus far into a transaction that immediately commits and initiating a new transaction (which might itself be elastic).

Closed and abstract nested transactions [12, 13] mitigate the cost of benign conflicts by restructuring large transactions, such that operations that are likely to suffer from such conflicts are moved into smaller, nested transactions. Nested transactions can then be rerun without re-executing their enclosing transaction.

The above techniques are all orthogonal to our approach. We expect that their incorporation into our synchronization algorithm should be seamless. We leave such integrations for future research.

User Annotations Udupa et al. [24] build a a parallelization system, which—based on programmer annotations—violates certain dependencies while preserving overall program functionality. This enables reordering of loop iterations, as well as tolerance of stale reads. The system can also infer which annotations are likely to improve performance through use of a test-driven framework.

The *early-release* [15] technique, developed by Herlihy et al., allows the programmer to specify object-specific semantics that the transaction can then utilize to shrink its read set in certain computation flows. An example of this is Figure 3.

Similarly to these works, JANUS has facilities for expressing consistency relaxations. The specification can either be granular and refer to specific shared objects, or it can be coarse and hold simultaneously for all the shared objects. JANUS also supports limited inference capabilities (cf. Section 5.3). The inferred annotations are sound, and cannot result in incorrect parallelization. This is in contrast to user-provided annotations, which both JANUS and the works above assume—rather than verify—to be correct.

9. Conclusion

We have presented JANUS, a speculative parallelization system that performs accurate conflict detection by testing entire sequences of operations—rather than individual operations—for conflicts. A key feature of JANUS is offline training: JANUS utilizes profiling data to build a specialized, sequence-based commutativity specification that can be queried efficiently at runtime.

Our evaluation of JANUS on five real-world benchmarks indicates that sequence-based detection is effective. JANUS achieves a speedup of up to 2.5x, where its instantiation with write-set-based detection fails to exploit the available parallelism, and even degrades performance, with 22x more false conflicts.

Acknowledgments

We thank Adam Morrison, Guy Gueta and the anonymous referees for their insightful feedback on earlier versions of the paper.

References

[1] The chord analysis framework. `http://code.google.com/p/jchord/`.

[2] The jfilesync utility. `http://jfilesync.sourceforge.net`.

[3] The jgrapht java graph library. `http://www.jgrapht.org`.

[4] Pmd java source code scanner. `http://pmd.sourceforge.net`.

[5] The sat4j sat solver. `http://www.sat4j.org/`.

[6] The sourceforge code repository. `http://sourceforge.net/`.

[7] The weka machine-learning library. `http://weka.sourceforge.net`.

[8] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, pages 757–762, 1966.

[9] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 364–376, 2003.

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

[11] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–107, 2009.

[12] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

[13] T. Harris and S. Stipic. Abstract nested transactions. The 2nd ACM SIGPLAN Workshop on Transactional Computing, 2007.

[14] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*. ACM, 2008.

[15] M. Herlihy, V. Luchangco, P. A. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[16] M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[17] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA*, pages 160–168, 2008.

[18] E. Koskinen, M. J. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, pages 19–30, 2010.

[19] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *PLDI*, 2011.

[20] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[21] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an stm. In *PPOPP*, pages 163–172, 2009.

[22] O. Tripp, R. Manevich, J. Field, and M. Sagiv. JANUS: Exploiting parallelism via hindsight. Technical report, Tel Aviv University, 2012.

[23] O. Tripp, G. Yorsh, J. Field, and M. Sagiv. Hawkeye: effective discovery of dataflow impediments to parallelization. In *OOPSLA*, pages 207–224, 2011.

[24] A. Udupa, K. Rajan, and W. Thies. Alter: exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 480–491, 2011.

[25] J. Whaley. Joeq: A virtual machine and compiler infrastructure. *Sci. Comput. Program.*, 57:339–356, 2005.