The IceCube approach to the reconciliation of divergent replicas

Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro
Microsoft Research
Cambridge, CB2 3NH, UK
{annemk,antr}@microsoft.com, marc.shapiro@acm.org

Peter Druschel*
Rice University
Houston, TX 77005, USA
druschel@cs.rice.edu

ABSTRACT

We describe a novel approach to log-based reconciliation called IceCube. It is general and is parameterised by application and object semantics. IceCube considers more flexible orderings and is designed to ease the burden of reconciliation on the application programmers. IceCube captures the static and dynamic reconciliation constraints between all pairs of actions, proposes schedules that satisfy the static constraints, and validates them against the dynamic constraints.

Preliminary experience indicates that strong static constraints successfully contain the potential combinatorial explosion of the simulation stage. With weaker static constraints, the system still finds good solutions in a reasonable time.

1. INTRODUCTION

Data replication is widely used in distributed systems for performance, availability and isolation. It enables access to shared information while disconnected (mobile computing) and to work independently (groupware). However, replication of mutable shared data inevitably raises the issue of consistency. Pessimistic consistency techniques require remote synchronisation upon each modification. This is detrimental in mobile systems, and is undesirable in groupware.

1.1 Reconciliation of diverging replicas

Alternatively a system may allow replicas to be updated independently and in a conflicting manner, and to diverge. Reconciliation is the activity of detecting, managing and resolving conflicts, in order to produce a new consistent value.

What constitutes a conflict and how to resolve it depends on semantics and on user intent. (One example is CVS [3], where non-overlapping writes conflict if and only if they occur in the same line of the same text file.) Accordingly, many existing reconcilers are restricted to a single data type, such source files [3], a file system [1, 8] or calendars. In contrast, IceCube aims to provide a general framework for reconciliation, parameterized to observe the specific semantics of a given shared data type, application, or user.

IceCube is log-based: the input to the reconciler is a common initial state and logs of actions that were performed on each replica. The reconciler merges the logs in some fashion, and replays the operations in the merged log against the initial state, yielding a reconciled, common final state. Logs provide the reconciler with the history of user actions; thus it can infer information about the users' intents and is therefore potentially more powerful.

In previous systems [4, 5, 10] the logs are merged according to some predetermined order, such as temporal order. These systems cannot exploit cases where a reordering of operations would avoid a conflict. IceCube attempts to find an ordering that minimizes conflicts, while observing object and application semantics and user intent. However, naively exploring all possible orderings suffers from a combinatorial explosion. Effectively pruning the space of orderings is one of the key design issues in IceCube.

1.2 Contributions

IceCube provides developers with a general but parameterisable framework for reconciliation. IceCube explicitly captures constraints between actions. A central feature is the distinction between static and dynamic constraints.[1] IceCube exploits static constraints in order to reduce the search space. A scheduling stage produces schedules that satisfy the static constraints. These schedules are then verified in a simulation stage, where actions are executed against a shadow copy of the state, to check for dynamic conflicts. Finally, a selection stage chooses among those schedules that did not exhibit dynamic conflicts.

*Work performed whilst on sabbatical at Microsoft Research, Cambridge, UK.

[1] Informally, a static constraint does not depend on the state of shared objects, whereas a dynamic constraint cannot be detected statically, i.e., without attempting to execute the actions.

Appears in Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001), 26-29 August 2001, Newport, Rhode Island (USA)

Developers can influence the outcome of reconciliation, either by providing local semantic information (in the form of pre- and post-conditions and the order method), or using local and global policy hooks.

This paper focuses on our approach to reconciliation at a single site, and we ignore the distributed aspects of the system.

The paper proceeds as follows. Section 2 presents the main concepts of IceCube and the system overview. Section 3 details the reconciliation process. We report on experiments with a collaborative jigsaw application in Section 4. We compare IceCube with related work in Section 5 before concluding.

2. ICECUBE OVERVIEW

One of the main powers of IceCube is the ability to use some predetermined order for the updated. This distinguishes IceCube from several other distributed systems. In order to demonstrate why this flexibility is important, let us consider two motivating examples.

As the first motivating example, consider two users A and B who collaboratively administer a computer system through two shared objects: the operating system, initially at Version 4, and the expense budget, initially containing £1000. User A upgrades the operating system, which automatically upgrades all installed drivers; realises that the new OS needs a new tape drive; and obtains a budget increase to cover the cost. User B buys and installs a printer and its driver within Version 4 of the operating system.

User A	User B
A1 Upgrade OS and drivers from v4 to v5.	B1 Buy printer, cost £400.
A2 Buy tape drive, £800.	B2 Install printer driver, OS v4.
A3 Obtain £1500 budget increase.	

Many collaborative tasks such as this one combine several applications and data types, which motivates a general-purpose system. However the system must take into account the semantics of objects; for instance the budget may not go negative. It is obvious that state-based reconciliation will not work with this example. Even a log-based system will fail if it tries to combine the logs in a pre-established order. For instance running log A before log B will fail because action B2 will find the OS in the wrong version. Running B before A will fail because the budget goes negative. Interleaving log A and B fails similarly. Instead the system should recognise both dependencies across logs (B2 must run before A1) and independencies inside a log (A3 may run before A1 and A2). Our IceCube implementation proposes solution A3, B1, B2, A1, A2, and recognises that other solutions (for instance B1, B2, A3, A2, A1) are statically equivalent and do not need to be evaluated.

As a second motivating example, consider a calendar application, with users A, B and C. As of Friday evening, A has no appointments on Monday morning, B has two free slots at 9:00 and 10:00, and C has appointments that fill all Monday morning. During the weekend the three users attempt to make appointments in their off-line calendars, such that A requests a one-hour appointment with B for as close to 9:00 as possible on Monday morning (app_{AB}), B requests a one-hour appointment with C for as close to 9:00 as possible on Monday morning (app_{BC}), and User C cancels their appointment at 9:00 on Monday morning ($free_C$).

The order in which these updates are applied is important. Assuming the users arrive at work at 8:55 Monday, there is only one ordering in which they can all be successfully applied: $free_C$, app_{BC} and then app_{AB}. Clearly, there is a dependency between $free_C$ and app_{BC}, since a free slot must be created for C if B is to make an appointment with C. If app_{AB} is performed before app_{BC} then B will be booked at 9:00, and C will be free only at 9:00, hence app_{BC} will fail.

In IceCube, the correct ordering will be determined, and the updates will be applied without generating any rejected appointments. However, in general the problem is that the number of potential orderings can be very large, and therefore it is essential to capture static conflicts that reduce the size of the search space.

2.1 System model

With IceCube, an application is either in the *isolated execution* phase or in the *reconciliation phase*. During isolated execution, a site executes its applications against a local replica of the shared objects, called the *object universe*. This brings the local object universe from some initial state to some *tentative* final state. Actions are recorded in a local *log*.

A *log* is an ordered set of actions. Although an isolated log is tentative, it has been successfully performed against the local object universe. Furthermore it satisfies some user intent, as well as correctness invariants. In that sense, an action log is correct. On the other hand, these constraints generally only impose a partial order on the actions in the log. In other words, some of the actions in the log can be reordered without violating correctness.

During the *reconciliation phase*, the logs of two or more replicas are merged to bring the replicas to a consistent state. In IceCube, reconciliation proceeds in three stages:

- When combining two logs, the scheduling stage considers all possible combinations of their actions, to generate replay *schedules.*[2] A *schedule* is a sequence of actions that satisfies the static constraints and is proposed to the next stage.
- The simulation stage then plays schedules against scratch replicas of the shared objects. A schedule that does not satisfy a *dynamic constraint* is aborted.
- The selection stage ranks and chooses among outcomes from the simulation.

[2] Actions are assumed deterministic. Replaying a log against the initial state results in the same final state.

Figure 1 depicts the architecture of the IceCube framework. To simplify exposition we present the stages as if they execute sequentially. In fact they run in a pipeline with various feedback loops, in order to provide better interactivity and faster response.

2.2 Actions

An action is the basic unit recorded in the log. We think of an action as an object composed of four parts:

- A *target*, identifying the shared object(s) that this action accesses.
- A *precondition*, a method with no side effects that returns a boolean.[3] If true, the operation proceeds, otherwise execution fails. A precondition is a dynamic constraint, often used to check that the state of the object universe at simulation time is compatible with the one observed during isolated execution (similarly to, but more flexibly than, a database lock).
- An *operation*, a method that may have side effects on the object universe. It returns a boolean to indicate post-condition success or failure, another dynamic constraint.
- Any amount of private data, collectively called a *tag*. The tag stores all the information about this action that will be used to evaluate static constraints involving this action. A tag often records the type of operation and its parameters.

Codes of the precondition and operation methods are opaque to IceCube. The tag is also opaque but will be accessed by the order method described later (in Section 2.4).

2.3 Constraints

Reconciliation should combine the initial logs in some way to produce a new log, which can be replayed to bring the object universe from its last common state to a new common state. Ideally, the reconciled log would contain all actions and satisfy object invariants and user intents. Instead of combining logs in some predetermined order, IceCube allows the application programmer to expose constraints between actions, and explores orderings that satisfy these constraints.

IceCube distinguishes between *static* and *dynamic* constraints. A static constraint can be evaluated without reference to the state of the object universe, i.e., is intrinsic to the nature and/or the tags of the two actions being compared. Static constraints restrict the search space of schedules in the scheduling stage. A dynamic constraint is checked both during the isolated execution stage and the simulation stage. It may refer to the current state of the object universe. A dynamic constraint is the return value either of the precondition or of the operation method (i.e., a post-condition) of an action; if either is false, execution is aborted.

A static constraint restricts *a priori* the possible orderings between two actions. The constraint relation between two

a \ b	write	read
write	maybe	unsafe
read	safe	safe

Figure 2: Read-write integer order (a, b), across logs

a \ b	increment	decrement
increment	safe	safe
decrement	maybe	safe

Figure 3: Counter integer order (a, b), across logs

arbitrary actions a and b has one of the values safe, maybe or unsafe, meaning a schedule containing a before b is respectively allowed, possible (modulo dynamic conflicts), or disallowed. In particular, if both constraint (a, b) = safe and constraint (b, a) = safe then a and b commute, and the scheduler is free to pick some arbitrary order. If two or more actions forms a cycle of unsafe constraints, they conflict statically, i.e., they can never all appear in the same schedule. In particular two mutually unsafe actions conflict with each other.

The scheduler compares every pair of actions, both across logs and within each log. The scheduler builds the static constraint relation from several sources: *(i)* the order of the logs, *(ii)* the identities of the target objects, and *(iii)* the order method (defined in the next section). If the actions' targets differ, they are assumed independent and commutative. If the actions come from different logs, IceCube constrains them according to the return value of each common target's order method. If both actions are from the same log, the order in which they appear in the log is safe (to reflect the end user's intent); the reverse order is constrained by order.[4] In summary:

$$\mathsf{constraint(a,b)} = \begin{cases} \mathsf{safe}\ \textit{if}\ \mathsf{a.target} \neq \mathsf{b.target} \\ \mathsf{safe}\ \textit{if}\ \mathsf{a}\ \textit{appears before}\ \mathsf{b}\ \textit{in the same log} \\ \mathsf{a.target.order(a,b)}\ \textit{otherwise} \end{cases}$$

2.4 The "order" method

For some object x, method x.order(a,b) compares two actions a and b that have x as a common target, returning safe, maybe or unsafe to signify that according to the semantics of x, ordering action a before b in a reconciled schedule is respectively allowed, possible (modulo dynamic conflicts), or disallowed. This provides the bridge between semantics and static constraints. Only local knowledge is needed, viz., a specification of the interactions between two given actions at object x. Accordingly, writing an order method is a relatively intuitive task.

Let us proceed by example. If some action u uses object x and action d deletes x, then u must appear before d. Accordingly we expect that for any shared object type T, T.order (u, d) = safe and T.order (d, u) = unsafe.

[3] In object-oriented programming terms, a method is a procedure in an object's interface.

[4] When an action targets multiple objects, the system calls each of their order in turn and returns the most constraining value.

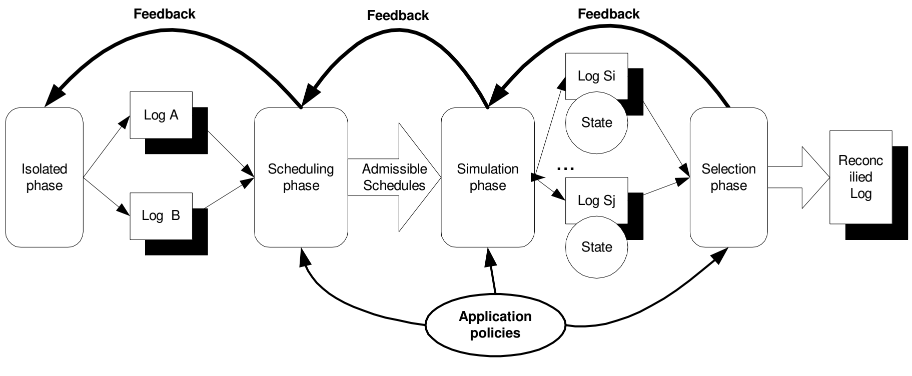

Figure 1: IceCube overview

a \ b	write	read
write	safe	unsafe
read	unsafe	safe

Figure 4: Read-write integer order (a, b), within log

a \ b	increment	decrement
increment	safe	safe
decrement	unsafe	safe

Figure 5: Counter integer order (a, b), within log

Consider now a shared integer variable R with the usual read-write semantics. When considering actions from different logs (i.e., independent work), the order values of Figure 2 avoid losing writes, but allow a read to be ordered before an unrelated write.

An integer representing a bank account or a budget has a different semantics: it may not become negative, and its methods are increment and decrement rather than read and write. The order method of Figure 3 orders increments before decrements; increments commute with one another, and decrements commute with one another subject to the dynamic constraint that the budget not become negative.

The result of the order method also depends on whether actions come from the same log or different logs. For instance, multiple reads of the same variable may be reordered within a log if they are not separated by a write to the variable; similarly, multiple writes to the same variable if they are not separated by a read of the variable. However, it is not safe to reorder a read with respect to a write of the same log, because that would change the value returned.

Figures 4 and 5 present the result of order for actions of the same log. Recall that the order provided by a log is safe by default; an in-log order method answers the following question: "given that the log contains action b before a, is it safe to swap them and execute a before b?"

An ordering is classified as safe (resp. unsafe) either because it is provably safe (resp. unsafe), or because application semantics make it desirable (resp. undesirable). Consider for instance two isolated users sharing files; one writes a file, while the other deletes that same file's parent directory. Formally it is safe to write then delete (the write does not influence the outcome of the delete), but not to delete then write (the write will fail). However the former ordering causes the first user's work to be lost. To avoid this, and contrary to mathematical intuition, the file system's order method marks the order write before delete as unsafe, and delete before write as maybe. This will trigger a dynamic failure and the user will be notified.

An order method can also support application-specific policies, as we will see in Section 4.

3. CONTROLLING THE RECONCILIATION ALGORITHMS

A key objective of the application programmer is to limit the space of possible schedules, in order to keep the simulation tractable. This is accomplished through static constraints, and through application-specific policies to dynamically truncate redundant portions of the search space. This section describes the algorithms used in the scheduling and simulation stages, and shows how the application programmer can control them.

3.1 Inputs to the scheduling and simulation stage

The inputs to scheduling and simulation consist of a set of actions A, relations $\mathcal{D}$ (dependence) and $\mathcal{I}$ (independence) on A, a starting state S_s, and a parameter H.

A is the set of actions to be reconciled. The *dependence* relation $\mathcal{D}$ is defined as follows. For actions $\mathsf{a}, \mathsf{b} \in A$, if $\mathsf{a}\mathcal{D}\mathsf{b}$ then action a must appear before (not necessarily immediately before) action b in any schedule that contains both a and b. $\mathcal{D}$ is reflexive and transitive.

The *independence* relation $\mathcal{I}$ is defined as follows. For actions $\mathsf{a}, \mathsf{b} \in A$, if $\mathsf{a}\mathcal{I}\mathsf{b}$ then the order a followed immediately by b is known (or highly likely) not to create precondition or execution failures in a schedule. $\mathcal{I}$ is not necessarily reflexive or transitive.

$\mathcal{I}$ and $\mathcal{D}$ are both empty by default. The constraint relation described in Section 2.3 maps to $\mathcal{I}$ and $\mathcal{D}$ as follows:

constraint(a, b)	$\mathcal{I}$	$\mathcal{D}$
safe	$\mathsf{a}\mathcal{I}\mathsf{b}$	-
maybe	-	-
unsafe	-	$\mathsf{b}\mathcal{D}\mathsf{a}$

The parameter H controls how the $\mathcal{I}$ relation affects the scheduling of actions for the simulation and allows application programmers to guide the scheduler into exploring those portions of the space of allowed sequences (according to the $\mathcal{D}$ relation) that are most likely to produce good solutions.

3.2 Dependence cycle analysis

Relation $\mathcal{D}$ may contain cycles, *i.e.* a set of actions $Cyc = \mathsf{c}_1, \mathsf{c}_2, ..., \mathsf{c}_n$, such that $\mathsf{c}_1\mathcal{D}\mathsf{c}_2, \mathsf{c}_2\mathcal{D}\mathsf{c}_3, ..., \mathsf{c}_n\mathcal{D}\mathsf{c}_1$. A schedule cannot both satisfy $\mathcal{D}$ and contain all actions of a cycle. We define a *cutset* S to be a subset of A, such that the removal of the actions in S and the associated edges from $\mathcal{D}$ leaves no cycles in $\mathcal{D}$. A *proper cutset* S is a cutset such that no proper subset of S is also a cutset.

The first step of the scheduler is to detect cycles in $\mathcal{D}$ and to generate all proper cutsets. The application then accepts one or more cutsets. For each one, the system then generates and simulates sequences, as described in the next section.

3.3 Scheduling

Now, we define how the $\mathcal{D}$ and $\mathcal{I}$ relations, and the parameter H, affect the scheduling of actions.

Consider an ordered prefix of already chosen actions $P = \mathsf{a}_1, \mathsf{a}_2, ..., \mathsf{a}_n$. S is the set of actions that can extend P by one action, according to $\mathcal{D}$. Recall that if both a and b appear in a schedule, and $\mathsf{a}\mathcal{D}\mathsf{b}$, then a must appear before b. Accordingly, an action b may be in S iff, either it does not depend by $\mathcal{D}$ on any other action, or that other action is already in P. Formally: $\forall \mathsf{b} \in S, \mathsf{a}\mathcal{D}\mathsf{b} \Rightarrow \mathsf{a} \in P$.

The scheduling of the next action to follow P then depends on the value of parameter H, which controls which heuristic is used to limit the size of the search space. Informally, when $H = All$, the scheduler generates all sequences that are consistent with the partial order defined by $\mathcal{D}$. When $H = Safe$, the scheduler generates sequences that use only safe orderings when choosing the next action to follow a given prefix, whenever such safe orderings exist. When $H = Strict$, the scheduler generates sequences that use only one of potentially multiple possible safe orderings, when choosing the next action to follow a given prefix.

A formal definition of H follows. Let C be a subset of S, such that $\forall \mathsf{c} \in C, \mathsf{a}_n\mathcal{I}\mathsf{c}$. That is, actions in C are known or highly likely not to cause a precondition or execution failure when immediately following P. Lastly, let B be a subset of S, such that $\forall \mathsf{b} \in B, \exists \mathsf{c} \in C, \mathsf{c}\mathcal{I}\mathsf{b}$.

$H = All$: The scheduler tries all actions in S as immediate successors to P. That is, $\mathcal{I}$ is ignored.

$H = Safe \wedge C \neq \emptyset$: The scheduler only tries actions in C as immediate successors to P. That is, if "safe" next actions exist, only those actions are tried.

$H = Safe \wedge C = \emptyset$: The scheduler tries all actions in S as immediate successors to P.

$H = Strict \wedge C \neq \emptyset$: The scheduler picks one action in C arbitrarily and tries only this action as a successor to P.

$H = Strict \wedge C = \emptyset$: The scheduler only tries actions in $S - B$ as immediate successors to P. This choice maximizes opportunities for the use of safe orderings when choosing subsequent actions.

3.4 Simulation

Simulation is interleaved with scheduling. The scheduler and simulator recursively explore all schedules that are consistent with $\mathcal{D}$ and $\mathcal{I}$, as described in the previous section.

A simulation step has the following inputs: *(i)* a prefix of already chosen and executed actions, *(ii)* the current state of the object universe (resulting from execution of the prefix from initial state S_s), and *(iii)* the next action to be simulated.

The action's precondition is evaluated within the current state. If true, the action is executed on a shadow copy of the state. If execution succeeds (i.e., there were no runtime errors and the postcondition is true), then the new action is appended to the current prefix. The resulting state becomes the input state for a recursive simulation of all sequences starting with the current prefix.

If the precondition fails, the application is notified and this branch of simulations aborts. If the execution fails, the application is notified, the shadow copy of the state is discarded and the current branch aborts.

3.5 Application policy hooks

For IceCube to be flexible and generic, application specific policies can be applied in various forms during the scheduling and simulation stage, as described in this section. First of all, applications may define an order function that enforces a particular heuristic. For example, keeping the log ordering for a given type of action as we'll see in Section 4.

Second, the application can influence the selection of cutsets. This can be used, for instance, to prioritise an action by not allowing it to be excluded from the reconciled log. For a given cutset, the application controls the order in which the schedules are explored.

During sequence generation and scheduling, the application can apply policies in several ways. Based on an intermediate state, the application may decide to abort the simulation of a prefix that is deemed not sufficiently promising, or it may inject additional static dependencies, conditional on the current prefix. If a precondition or execution failure occurs, the application is provided with the prefix and state causing the failure. The application may analyse the state and derive additional information about the causes of the failure.

Finally, the application is called whenever a complete schedule that include all actions in A minus the cutset is found. The application selects among multiple successful outcomes e.g. using an application-specific cost function.

4. APPLICATION EXPERIENCE: COLLABORATIVE JIGSAW

This section reports on our experience with a multi-user jigsaw puzzle.[5] This application (Figure 6) was chosen as representative of a collaborative workload.

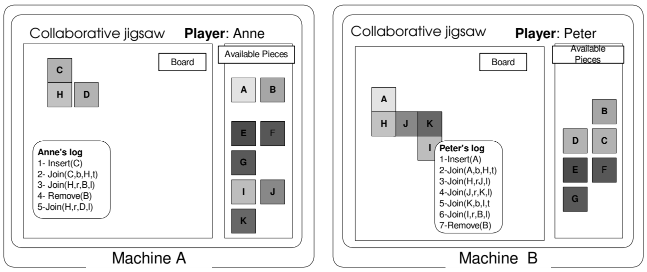

Figure 6: Two users' view of Jigsaw

4.1 Application description and semantics

A game state, *Jigsaw*, is composed a fixed set of pieces, some on the *board*, the others available. A state is *correct* if every piece on the board so far matches its correct position in the final picture. Every participant starts with an empty board.

A completed board contains $n \times m$ pieces $P_0 ... P_{n\times m-1}$. Each piece has a square shape with 4 edges: bottom, top, left and right. Once the board has been initialised with a single *insert*, a participant plays in isolation, executing a series of *join* and *remove* actions:

join(P_i, e_i, P_j, e_j)
: *Target*: P_i, P_j
Precondition: *(i)* The board is not empty, and *(ii)* either P_i or P_j is available (but not both), and *(iii)* edge e_i of P_i and edge e_j of P_j are not already taken.
Operation: Move the available piece onto the board, join edges e_i and e_j of pieces P_i and P_j.
Tag: records operation type and parameters: join; $P_i, e_i; P_j, e_j$.

***remove** (P_i)*
: *Target*: P_i
Precondition: P_i is on the board.
Operation: Moves P_i from the board, make it available.
Tag: remove; P_i.

4.2 Experiments

We considered four different cases, varying the nature of the static constraints. In the first case we used "semantic" constraints, i.e., an order method designed to reflect the semantics of the objects involved (similarly to Section 2.4). For reasons that will become apparent, the three other cases use "application policy" constraints that do not reflect the semantics directly but are designed to orient the simulation towards promising schedules.

Case 1: The semantic constraints represent the rules of the game, and laws of physics such as "two different pieces can't join the same edge of the same other piece". The corresponding order method is given in Figures 7 and 8.

[5]A game where players are presented with pieces of a picture and reconstitute the whole picture.

	join (P_i, e_i, P_j, e_j)	remove (P_f)
join (P_k, e_k, P_l, e_l)	maybe if physically possible; unsafe otherwise	maybe
remove (P_m)	unsafe if $m = i$ or $m = j$; maybe otherwise	maybe if $m \neq f$; unsafe otherwise

Figure 7: Jigsaw game order: same log, semantic constraints

	join (P_i, e_i, P_j, e_j)	remove (P_f)
join (P_k, e_k, P_l, e_l)	maybe if physically possible; unsafe otherwise	unsafe if $k = f$ or $l = f$; maybe otherwise
remove (P_m)	unsafe if $m = i$ or $m = j$; maybe otherwise	maybe if $m \neq f$; unsafe otherwise

Figure 8: Jigsaw game order: across logs, semantic constraints

Case 2 uses a policy constraint that preserves each player's log order. Thus: for two actions a and b, order(b, a)= unsafe if a precedes b in the same log.

Case 3 keeps the log ordering for join actions only. This policy weakens the previous one, allowing removes to be scheduled at any place. For two actions a and b, order(b, a)= unsafe if a precedes b in the same log, and a and b are joins.

Case 4 applies the preference order aIb between join actions a and b having one piece in common. This policy favours uninterrupted strings of adjacent joins.

Within each case, several user scenarios are available. In scenario U1 a participant joins correct pieces, left to right, row by row downwards, starting from square 0. U2 is symmetric to U1: right to left and upwards, starting from the last square. In scenario U3 a participant executes a random sequence of correct and incorrect joins and removes (strongly biased towards correct moves) starting from square 0. We experimented two variants: *(i)* one user plays U1 and the other U2, *(ii)* one user plays U1 and the other U3.

Furthermore we varied the size of the board (up to 10×10) and the number of actions in each scenario (up to the maximum number of pieces).

Finally we varied the heuristics parameter H.

For space reasons, we provide only a limited number of representative measurements here.

4.3 Observations

We compared the reconciliation results according to different criteria: *(i)* the number of actions in the schedule, *(ii)* the number of pieces in the reconciled state, and *(iii)* the number of correct pieces.

Case 1 is based on the semantic constraints. With a board size of 4×4, reconciliation and simulation of a 20-actions game produces the best solution with respect to all the comparison criteria.

In this example, semantic constraints ensure immediate convergence. This is the preferred approach when logs are "clean," i.e., contain no redundant actions. If not spurious conflicts appear; consider for instance the case where one player adds then removes some piece, while another user concurrently inserts the same piece: this is flagged as a static conflict, even though the piece is dynamically available. Imposing a clean log either imposes unnatural restrictions on the interactive user, who is not allowed to change his mind, or assumes a mechanism to clean the log after the fact.

Cases 2 to 4 Because of the questions around semantic constraints, we tried weaker static constraints. We observe that, in the absence of any static constraints, simulation runs into combinatorial explosion and does not terminate in a reasonable time. Therefore the following experiments explored the application policy constraints, as described in Section 4.2. They are found to partially solve the problem.

Consider for instance a game where the first player puts 7 pieces in scenario U1, and the second puts down 12 pieces in scenario U2, on the 4×4 board.

For Case 2, when $H = \mathit{Strict}$, reconciliation provides two solutions, which are equivalent to log 1 and log 2 alone, respectively. When $H = \mathit{Safe}$ the result is the same; Cases 3 and 4 also give the same result, independently of the value of H. For Case 2, when $H = \mathit{All}$ the reconciler finds the optimal solution, i.e., where all 16 pieces are correctly placed. In this case, the simulator finds the optimal solution after two sequences, in 0,11 s, after which it continues to run through all possible 38,102 schedules. This would be appropriate if the user has immediate interactive feedback.

In a game where the second player follows scenario U3, we observe in Cases 3 and 4 occasional reorderings that provide better solutions than in Case 2 (which disallows reorderings).

Policy constraints do not always ensure convergence. As the size of the input logs increases, the stronger policies tend to over-constrain the system and no solution is found; the weaker policies do not terminate within the (arbitrary) limit of 100,000 simulations.

Finally we measure the overhead of static constraints. In the absence of static constraints, a simulation of 10,000 schedules is 0.781s. In Case 2 the same number of schedules is simulated in 2.294s, three times longer. Simulation times are proportional to the number of simulated schedules. For instance 100,000 simulations without static constraints terminate in in 7.7s.

4.4 Discussion

The experiments confirm that strong static constraints are necessary to limit combinatorial explosion in the simulation stage, but the results are very sensitive to the choice of constraints. Both semantic constraints and policy constraints were considered.

Semantic constraints are desirable but impose "clean" logs. If more than one action in the same log updates the same object, spurious conflicts may occur. For instance, consider some action A that semantically conflicts with B. The interactive user will expect that cancelling out A with some compensating action $\tilde{A}$ will remove the conflict. On the contrary, the $A - B$ and the $\tilde{A} - B$ pairs both flag a static conflict.

If the application is not going to impose unreasonable constraints on the interactive user, one possible solution is *log cleaning* [2, 4], i.e., to combine several actions from a same log targetting a same object into a single one. For instance in the jigsaw game, the sequence join (P_1, top, P_2, bottom), remove (P_2), join (P_1, top, P_2, bottom) would be reduced to join (P_1, top, P_2, bottom).

5. RELATED WORK

Isolated execution is related to optimistic concurrency control [9], which allows concurrent transactions to access shared data, checking for dependencies only when transactions commit. In particular Herlihy [7] defines precisely what it means for a schedule to be correct in the context of a semantic concurrency relation similar to our order. In contrast, the focus of this paper is reconciliation, i.e., finding alternatives when a schedule fails.

The work of Phatak and Badrinath [11] presents similarities with IceCube although our ideas were developed independently. They present an incremental algorithm, based on optimistic concurrency control in a multi-version database, for incorporating disconnected transactions into a schedule. It inserts each such transaction into the schedule at an optimal position, such that the combined schedule respects a weakened snapshot isolation condition. One key difference is that their preconditions are based purely on read-sets and write-sets, whereas in IceCube preconditions take into account data semantics. Another is that they assume transactions are independent, whereas IceCube supports dependencies between actions. Finally, Phatak and Badrinath's algorithm lacks a scheduling phase, which we found essential to fight combinatorial explosion. It is not clear whether their algorithm has been implemented.

Recent work by Ramsey and Csirmaz [12] proposes an algebraic approach to file synchronisation, a restricted instance of the general reconciliation problem. Operations on files are carefully crafted to make them almost entirely independent and idempotent. The only dependencies are between an object (file or directory) and the existence of its ancestor directories. A log is assumed clean, i.e., it contains no more than one operation affecting a given object. This allows them to define a canonical ordering between operations such that reconciliation has a unique, static solution: non-commutative operations appear in their natural order, and commutative operations are ordered arbitrarily but consistently. This is a very elegant approach, but it is not clear how it will fare in practice.

Schwarz et al. [13] describe lock compatibility tables that capture interactions between concurrent transactions, in order to increase concurrency. A compatibility table captures static conflicts, similarly to our order method. However, order captures not just compatibility but also ordering information. Moreover, IceCube's maybe points to possible conflicts to be checked at simulation time using the dynamic preconditions. IceCube is designed to facilitate reconciliation, not to increase transaction throughput.

IceCube's logs are modeled after Bayou [10], where a logged action contains a precondition (called a dependency check) and code that performs the operation. Bayou also provides for an alternative code path called mergeproc, invoked when the precondition fails (i.e. in case of a conflict), which is supposed to resolve the conflict. Bayou may reorder actions several times; the committed order is consistent with the real times at actions were accepted at their respective "home" servers; this ordering is arbitrary from a client's perspective. IceCube extends these ideas, and attempts to provide a best ordering. IceCube also captures more static information, using the order method. IceCube would be able to schedule actions automatically without conflict in some cases where Bayou spuriously invokes mergeproc. Thus IceCube arguably reduces the burden on the application programmer.

Reconciliation needs to compensate for the difference between an operation performed by an isolated user in the context of its local view of the shared object universe, and performing the same operation in the context of the reconciled state of the object universe, in a manner that preserves the user's intent. For instance, a text editing application might designate edits by the position of the affected characters in the text—but concurrent edits scheduled earlier by reconciliation might change that numbering, or even invalidate or obviate the need for the edit. Accordingly, arguments need to be translated to make sense in the new context, viz., character numbers remapped. This translation, called Operational Transformation, is surprisingly complex [14]. In general, performing such transformations correctly requires that the system capture and observe as much information about the users' intent as possible and is arguably the key problem in reconcilation.

6. CONCLUSION

We presented IceCube, a general-purpose log-based reconciliation system parameterised by object and application semantics. It differs from previous work in that the ordering of the reconciled log is computed flexibly, and that the programmer's contribution to reconciliation is relatively simple. Although two actions from different logs are causally independent, IceCube discovers implicit (data and semantic) dependencies and constrains schedules accordingly. Although two actions from a same log have been executed in sequence, IceCube discovers when they are independent and relaxies schedules accordingly. Static constraints are crucial to controlling combinatorial explosion.

We have reported on preliminary experiments, which are promising where there are strong static constraints. Otherwise, appropriate policies help find good-quality answers in an acceptable amount of time.

The current status suggests several lines of future work. We need to experiment with larger and more realistic examples, based on real-life situations. Although the scheduling stage and the application policies reduce the search space considerably, additional techniques to further focus and narrow the search may be necessary. In particular, we are currently investigating the use of constraint programming methods in IceCube [6]. Also, we envisage to use the causality information encoded in the order method to identify schedules that will fail identically. Finally, the action-based style of programming is unfamiliar, but could probably be made easier with appropriate programming language support.

7. ACKNOWLEDGMENTS

We would like to thank Silvano Dal Zilio and François Fages for their helpful suggestions.

8. REFERENCES

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Int. Conf. on Mobile Comp. and Netw. (MobiCom '98)*. ACM/IEEE, Oct. 1998. http://www.cis.upenn.edu/~bcpierce/papers/snc-mobicom.ps.

[2] M. Berger, A. Schill, and G. Volksen. Supporting autonomous work and reintegration in collaborative systems. In W. Conen and G. Neumann, editors, *Coordination Technology for Collaborative Applications — Organizations, Processes, and Agents*, pages 177–198, New York, 1998. Springer. http://wwwrn.inf.tu-dresden.de/lsrn/Ps-Files/autonomous.ps.

[3] P. Cederqvist et al. Version management with CVS, 1992.

[4] O. Dedieu. *Réplication optimiste pour les applications collaboratives asynchrones.* PhD thesis, Université de Marne-la-Vallée, Marne-la-Vallée (France), Sept. 2000. http://www-sor.inria.fr/~dedieu/these.html.

[5] W. K. Edwards, E. D. Mynatt, K. Pertersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. Designing and implementing asynchonous applications with Bayou. In *Symp. on User Int. Software and Tech.*, Banff Alberta (Canada), Oct. 1997.

[6] F. Fages. A constraint programming approach to log-based reconciliation problems for nomadic applications. In *6th Annual W. of the ERCIM Working Group on Constraints*, Prague (Czech Republic), June 2001.

[7] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *Commun. ACM*, 15(1):96–124, 1990.

[8] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. of the Usenix Winter 1995*, Jan. 1995.

[9] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), June 1981.

[10] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. Symp. on Operating Systems Principles (SOSP-16)*, pages 288–301, Saint Malo, Oct. 1997. ACM SIGOPS. http://www.parc.xerox.com/csl/projects/bayou/.

[11] S. H. Phatak and B. R. Badrinath. Conflict resolution and reconcilation in disconnected databases. In *2nd Int. W. Mobility in Databases and Dist. Sys. (MDDS'99)*, Florence (Italy), Aug.–Sept. 1999. http://www.cs.rutgers.edu/~badri/papers/reconciliation.pdf.

[12] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. Technical Report TR-05-01, Harvard University Dept. of Computer Science, Cambridge MA (USA), May 2001.

[13] P. M. Schwartz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, Aug. 1984.

[14] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Conf. on Comp.-Supported Cooperative Work (CSCW)*, page 59, Seattle WA (USA), Nov. 1998. http://www.acm.org/pubs/articles/proceedings/cscw/289444/p59-sun/p59-su% n.pdf.