Optimistic Parallelization of Communicating Sequential Processes

David F. Bacon, University of California, Berkeley and IBM T.J. Watson Research Center
Robert E. Strom, IBM T.J. Watson Research Center
dfb@cs.berkeley.edu, strom@watson.ibm.com

Abstract

We present a transparent program transformation which converts a sequential execution of $S_1; S_2$ by a process in a multiprocess environment into an optimistic parallel execution of S_1 and S_2.

Such a transformation is valuable in the case where S_1 and S_2 cannot be parallelized by static analysis either because S_2 reads a value from S_1 or because S_1 and S_2 each interact with an external process. The optimistic transformation works under a weaker set of conditions: (1) if the value S_2 reads from S_1 can usually, but not always, be correctly guessed ahead of time, and (2) if S_1 and S_2 interact with an external process, conflicts which violate the ordering of S_1 and S_2 are possible but rare. Practical applications of this approach include executing the likely outcome of a test in parallel with making the test, and converting sequences of calls into streams of asynchronous sends.

We analyze the problem using the framework of *guarded computations*, in which each computation is tagged with the set of guesses on which it depends. We present an algorithm for managing communications, thread creation, committing, and aborting in the transformed program. We contrast our approach with related work.

1 Introduction

The problem of how to parallelize sequential processes has been extensively studied. Existing approaches use static analysis [2, 5] to prove that two sequential code segments do not interfere. If this can be proven, then both segments are run in parallel; if not, they must be run sequentially.

Proc. of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP 1991), Williamsburg, Virginia. SIGPLAN Notices Vol. 26, No. 7, July 1991, pp. 155–166.

This technique is rarely applicable to a process in a distributed system because it may communicate with other processes not known at compile-time, and because the non-determinism introduced by communication makes non-interference difficult to prove.

However, many circumstances arise in distributed systems where non-interference can be guessed with high probability, even though it cannot be statically guaranteed. For instance, suppose process X is an application and process Y a window manager. Process Y exports a service named `PutLine`. It takes a single line per call and returns an indication of whether the line was displayed successfully. Process X repeatedly calls `PutLine`, passing it successive output lines until all output has been delivered or until it receives an unsuccessful return code. In a straightforward implementation, process X waits after each call for a return from Y before making the next call. If Y is remote, and communication delays are long relative to the speed of computation, such an implementation is inefficient. Process X cannot be parallelized by conventional static analysis because of the dependency between each call and the return code from the previous call.

We can improve performance by transforming process X into an *optimistic* execution containing two parallel threads. The first thread makes the call and waits for the return, while the second executes the continuation after the call, assuming that the call completed with a successful return code. If the continuation makes further calls to `PutLine`, the process is once again split, with the effect that a series of two-way calls is converted into a series of one-way sends. This transformation is termed *call streaming*. It is extremely valuable when bandwidth is high but round-trip delays are long.

Every optimistic protocol relies on a guess. In the case of call streaming, the guess is that the first thread's call (1) will be serviced before the second thread's call, and (2) will return successfully. Some systems, such as the X-window protocol, trade off correctness for performance by providing an asynchronous send-based interface, and requiring the user to handle asynchronous notification of errors. In contrast, in our approach we abort the second thread and undo all its side effects if either of our guesses was wrong. This way our results are correct whether we guess right or wrong, and provided we usually guess right, we still obtain a performance improvement.

This paper presents an optimistic protocol for transforming sequential code into parallel code. We use the call streaming transformation as a running example. However, our approach is applicable to parallelizing arbitrary sequential constructs $S_1; S_2$ in the case where

- S_2 depends upon S_1. Static analysis will indicate that the two process fragments are not parallelizable, either because for some execution path S_2 reads a value from S_1, or else because S_1 and S_2 each interact with some external process, such as a server in a distributed system.
- There is a way to guess the result of S_1 with a high probability of success.

Our program transformation is *transparent*. That is, the programmer makes no change to the source code, and sees no change in the program behavior, except for execution speed. The transformed computation completes earlier whenever the guess is correct, and later whenever the guess is incorrect. If a bad guess is made, the program still runs correctly, but the average performance will be worse because of excessive rollbacks.

Our solution is based on earlier work on Optimistic Transformations [10] and is similar to Optimistic Recovery [9]: we guess that conflicts between serial segments of a process will not occur. When a conflict occurs, we roll back the processes that depend on the incorrect guess and restart them, running the computation in the original serial order.

2 Formulation and Example

We assume a high-level model in which independent sequential processes communicate by message passing or by making inter-process calls, as in CSP [3], Ada [12], or Hermes [8]. A feasible target environment is the Mach operating system [1], which supports both parallelism and distribution.

Each process is written as a sequential program. However, the processes are compiled and instantiated separately. This means that there is no *a priori* ordering of events between processes. Instead, there is a partial order of events between processes which is dynamically determined by their communication. These assumptions are typical of distributed systems.

Within a process there is a sequence of program segments $S_1; S_2$. Both S_1 and S_2 may include interactions with other processes in the system. We assume the that there is some mechanism by which the compiler is told that it is desirable to parallelize S_1 and S_2. This mechanism could be programmer supplied pragmas, run-time profiling, static analysis, or a combination of these methods.

We further assume that either S_2 uses no values defined in S_1 (the only "guess" is that S_1 terminates without interfering with S_2), or that the compiler has been told what to guess for values defined in S_1 and used in S_2. These values are said to be *passed* from S_1 to S_2.

Our transformation generates a parallel execution of S_1 and S_2. To be correct, our implementation must preserve the traces of the original program. A trace is a collection of observable events. In this implementation, the observable events are the messages sent and received by all computations except those which are "aborted" by our mechanism. We preserve both the data values of each input and output event, and the "happens before" relationship [7], which we denote by "$\rightarrow$", between these events. Specifically, if e_1 is an event in the execution of S_1 and e_2 is an event in the execution of S_2, then we must preserve $e_1 \rightarrow e_2$.

We illustrate our transformation with a somewhat more complex example of call streaming than we gave in the introduction. Suppose that process X is a client containing two segments S_1 and S_2. S_1 is an `Update` call to a database server (process Y), which actually writes the value by calling process Z, the network filesystem server. S_2 then makes a direct call to `Write` to the filesystem server. The only value passed from S_1 to S_2 is the variable `OK`, which indicates whether the call successfully updated the database. The compiler has been told that it is desirable to parallelize S_1 and S_2, and that `OK` is usually `true`.

Figure 1 illustrates the code of the original program segment. Figure 2 illustrates a typical execution in an unoptimized implementation, using a time-line diagram.

```
/* S1 */ OK = Update(Item,Value); /* call Y */
/* S2 */ if OK
            Write(File,"Did it"); /* call Z */
```

Figure 1: Program Segment to Be Parallelized

There are three obstacles to parallelizing S_1 and S_2: First, there is a data dependency on the boolean value `OK` between S_1 and S_2. Second, processes X, Y, and Z are independent of one another and inter-procedural analysis across the RPC calls is not possible. Finally, depending on communication delays, X's call to Z could arrive before Y's call to Z, which clearly violates the happens-before relationship.

We will overcome these problems by tracking dependencies dynamically, by exploiting the guess that `OK` is `true`, and by guessing that the messages arrive at Z in the correct order. Given these assumptions, we can transform X into two parallel threads: one to execute up to the end of S_1, and one to execute S_2 and its continuation.

Figure 3 illustrates a successful (non-aborting) execution of the transformed process X. The two threads are overlapped in real time, whereas they were sequential in Figure 2. The dotted arrow pointing backwards in real time indicates that the last event of the execution of S_1 logically precedes the first event in the execution of S_2; the labels in braces are used for dynamic dependency tracking and will be explained below. The happens-before relationship is determined by: (1) the physical ordering of events within each thread, (2) the logical ordering between the two threads depicted by the dotted line, and (3) the causal ordering between send events and receive events. The execution in Figure 3 generates the same events in the same order as the original execution in Figure 2, and is therefore a correct implementation.

If the call to `Update` returns `false` or fails to return at all, we call this a *value fault*, and the parallelized (right-hand) thread is aborted.

The parallelized thread will also be aborted if an execution such as the one in Figure 4 occurs. In this case, the call from X was received by process Z before the call from Y. This contradicts the logical order in which the return from Z to Y precedes the return from Y to

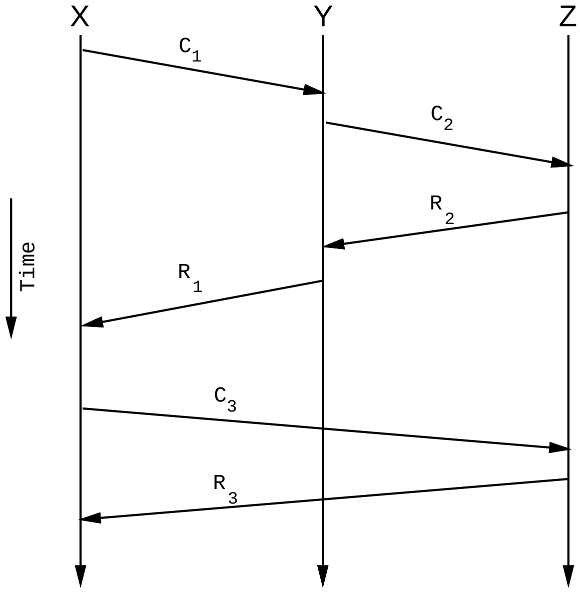

Figure 2: No Call Streaming. Processes X, Y, and Z communicate via calls and returns (labeled C and R).

X, which precedes X's call to Z. An examination of the diagram shows that there is a cycle in the "happens before" relationship. Such a cycle is a violation of causality, and is called a *time fault*.

3 Optimistic Execution

The example above provides the motivation for optimistic parallelization and gives a rough idea of our approach. We now provide the theoretical framework in which we solve the problem and show how we use this framework to create an optimistic algorithm.

3.1 Optimistic Transformations

Our solution is based on applying the theory of Optimistic Transformations [10] to the case of time faults and value faults. Optimistic Transformations is a framework for deriving optimistic distributed protocols from conservative protocols. In this framework, a process may choose at some point to make a *guess*, and initiate a computation based on this guess. For each guess, there is some stable predicate p_n which confirms the guess — that is, if p_n is true, then the computation x_n based on this guess is correct. In this case, p_n is said to be the *commit guard* of x_n. At any given time, the predicate

p_n is either known to be true, known to be false, or *in doubt*. While p_n is in doubt, there is a live computation to compute p_n. Therefore, any commit guard predicate is eventually discovered to be either true or false.

At any given time, a computation x_n is associated with a (possibly empty) set of identifiers of in-doubt predicates $\{p_1, \ldots, p_k\}$. This set is called the *commit guard set*. The commit guard predicate of x_n is the conjunction $\bigwedge_{i=1}^{k} p_i$. If the commit guard set of a computation is empty then the commit guard predicate is vacuously `true`—this means that the validity of the computation is independent of any guesses. Such a computation is said to be *committed*. A computation with a non-empty commit guard set is called *optimistic*. An optimistic computation eventually either commits or else aborts by rolling back to an earlier state.

For each in-doubt predicate p_i, there must exist a process which will eventually compute the truth or falsehood of p_i. The predicate is said to *commit* when it resolves to true, and *abort* when it resolves to false. When an in-doubt predicate p_i is committed or aborted, all computations containing p_i in their commit guard set are notified, and must abort. If a predicate in a computation's commit guard set commits, that predicate is removed from the set. When the commit guard set becomes empty, the computation commits.

Each message carries with it a tag containing the commit guard set of the computation which sent the message. When a computation receives such a tagged message, it acquires a new commit guard set consisting of the union of its previous commit guard set and the commit guard set of the message. In this way, commit guards track the causal dependencies between computations.

Messages to or from processes that can not be rolled back, such as workstation displays, printers, and systems not participating in our protocol, are called *external messages*. External messages sent by a guarded computation must be buffered, since we do not allow external observers to see possibly incorrect outputs. When a computation commits, it releases its external messages and discards any state it created for purposes of rolling back. When a computation aborts, it discards its external messages and rolls back. External input messages always have an empty guard set, since they are never aborted.

To implement optimistic transformations, the compiler generates the guarded computations and the computations which check the guesses. The run-time environment encodes each guess, maintains a commit guard for each current in-doubt computation, propagates commit guards on messages exchanged between processes, distributes *control messages* notifying other processes that a guess has committed or aborted, and maintains the ability to roll back state.

To prepare for rollback, a process may take a state checkpoint at each point prior to acquiring a new commit guard predicate. This approach is used in Time Warp [4]. Alternatively, a process may take less frequent checkpoints, and log input messages, restoring the state by resuming from the checkpoint and replaying logged messages. This approach is used in Optimistic Recovery. A process may also rollback by *inverting* the steps of its computation. The particular technique used for rollback is a performance tuning decision and does not affect the correctness of the transformation.

3.2 Algorithm Description

We now apply the above framework to create an algorithm for Optimistic Parallelization.

When we run S_2 without waiting for S_1 to complete, we are making two optimistic guesses:

- no value fault occurs: S_1 terminates, and passes the guessed values to S_2, and
- no time-fault occurs. For each time fault, there is a causal chain of events $e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_k$ where e_0 is in S_2 and e_k in S_1. The existence of such a chain implies that e_0 (in S_2) precedes e_k (in S_1), which contradicts the logical semantics $S_1; S_2$.

These guesses constitute the commit guard predicate associated with the parallel execution of S_2. If S_1 completes without violating either of these guesses, then the predicate is *committed*; otherwise it is *aborted*.

If the predicate is committed, execution of S_2 proceeds without the possibility of rollback (unless there are other commit guard predicates). A control message called a *commit message* is sent to other processes which depend upon the predicate which committed; this may enable them to commit as well. If the predicate is aborted, S_2 is rolled back and re-executed. Additionally, all its side effects are undone. A control message called an *abort message* is sent to other processes which depend upon the predicate which aborted; this may cause further rollbacks.

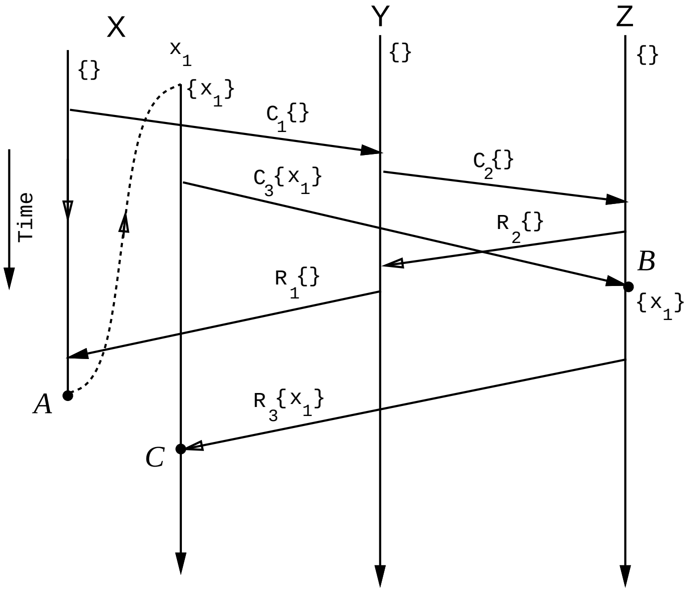

Figure 3: Successful Optimistic Call Streaming. Commit guard sets are indicated next to the process time lines and on the message identifiers, and points of interest are labelled A, B, etc. The dashed line indicates that A precedes x_1 in logical time.

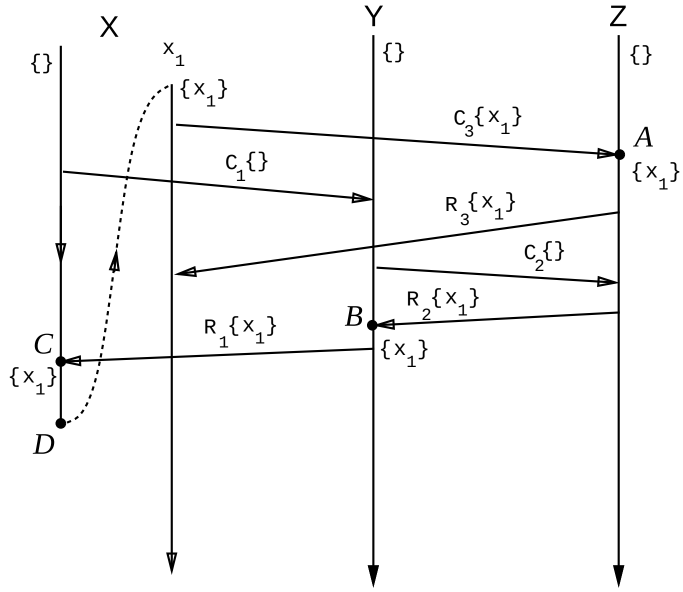

Figure 4: Aborted Optimistic Call Streaming. x_1 is aborted at D since $\{x_1\} \rightarrow \{x_1\}$, indicating a cycle in the happens-before relationship. The results are shown in Figure 5. The cycle includes C_3, R_2, R_1, and the dashed line.

Provided that the overhead of our method is small and the relative frequency of aborts is sufficiently low, we will achieve a performance gain by parallelizing S_1 and S_2.

Let $S_0; S_1; S_2$ be the code of a sequential process X which is to be transformed by running S_1 and S_2 in parallel. We call the boundary between S_0 and S_1 the *fork point* and the boundary between S_1 and S_2 the *join point*. Without loss of generality, assume that S_1 does not itself contain a computation which is being parallelized. Let $\{v_i\}$ be the variables passed from S_1 to S_2 and let $\{b_i\}$ be the values guessed for $\{v_i\}$. The $\{b_i\}$ depend only on the state at the fork point.

When execution reaches the fork point, the transformed program executes a *fork*, splitting the computation into two parallel threads. By convention, the left thread executes S_1; the right thread executes S_2. The left thread has the same commit guard set as S_0. The right thread has a commit guard set containing an additional predicate which we shall label x_1. The name x_1 will denote two things: (1) as an *event*, x_1 denotes the join point between computations S_1 and S_2 in process X; (2) as a commit guard predicate, x_1 denotes the commit guard predicate asserting that the left thread will complete with no value fault and with no time fault. It is possible that the right thread contains further fork points, in which case there will be a right-branching forking structure. In general, x_n will name the n^{th} join point event in X and will encode the predicate that the n^{th} forked computation will complete without a value fault or time fault.

The two threads will each run on a separate copy of the state, except that in the right thread, we start by setting each v_i to the guessed value b_i. If there is no antidependency between S_1 and S_2 — that is, there is no variable read by S_1 and overwritten by S_2 — then the copy is unnecessary. This is the case in our call streaming example.

After the fork, the execution of S_1 and S_2 proceed in parallel, but the computation of S_2 in the right thread is guarded by x_1. Any messages sent from the right thread will be annotated with the commit guard name x_1. Any process receiving such messages will inherit the commit guard x_1. In Figure 3, Y, Z, and the left thread of X have an empty commit guard set. The right thread of X is guarded by $\{x_1\}$, and when it sends C_3, this commit guard set is appended to the message; when Z receives C_3 at point B, its commit guard set is changed to $\{x_1\}$. This guard is also sent with R_3, but this does not change the guard at point C.

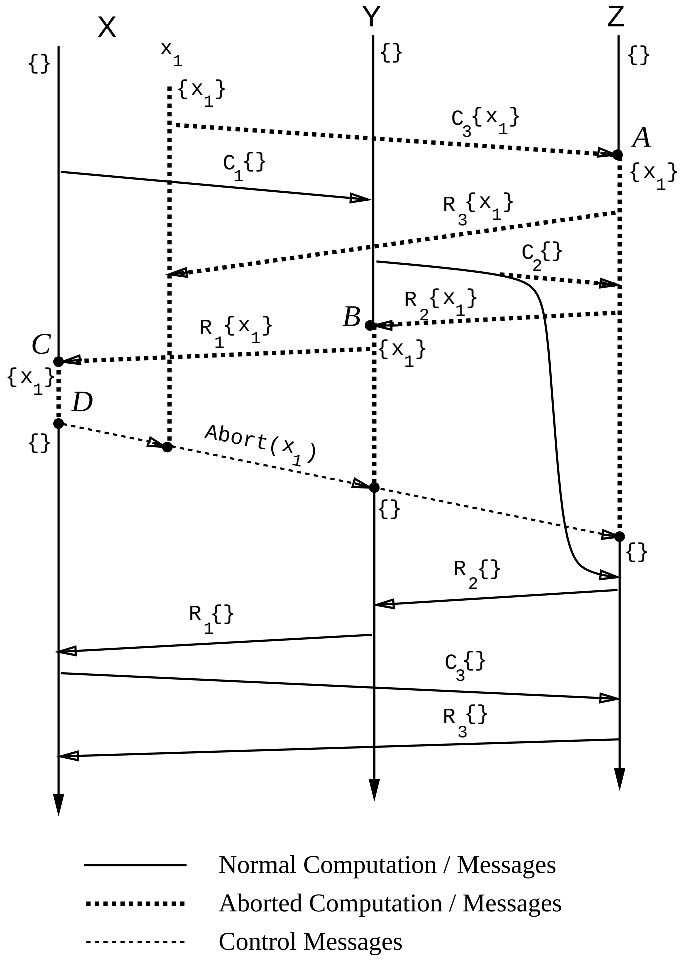

Figure 5: Abort and Re-execution. Note that process Z must re-read message C_2 after rolling back.

When the left thread completes, a check is made that the expected values $\{b_i\}$ were assigned to $\{v_i\}$. If the check fails, predicate x_1 is aborted due to a value fault. The right thread and any causal dependents are rolled back immediately, and an abort message is propagated to any processes which depend upon x_1. In Figure 3, this would mean rolling Z back to point B and aborting the right thread of X.

A timeout of implementation-defined duration is set at the beginning of the fork. The timeout guarantees that predicate x_1 aborts in case S_1 diverges. If the timeout expires, the left thread continues, but predicate x_1 is aborted, and a value fault due to non-termination is assumed.

If the left thread terminates with no value fault, we check for a time fault by examining the commit guard set at the termination point. Every time fault includes at least one dependency which goes backwards in real time—that is, one resulting from a fork. If S_1 terminates with an empty commit guard set, then there are no uncommitted forks in its causal past, and x_1 can be committed. This is the case at point A in Figure 3: R_1 has an empty guard set, and so the left thread terminates with an empty guard set and commits.

If the guard set at the end of the left thread is non-empty, there may be a time fault. Recall that the commit guard set includes all the uncommitted guards in the causal past. If x_1 belongs to that set, then we know there is a time fault, because this means that S_1's execution causally follows the S_2 thread. This is the case at point D in Figure 4: because C_3 arrived at Z before C_2, x_1 is in the guard set of R_2. This causes x_1 to be added to the guard set of Y at point B, and R_1 then propagates it to the left thread of X at point C. As a result, x_1 is in the causal past of point D. But since point D logically precedes the beginning of the right thread, we have $x_1 \rightarrow x_1$, meaning that a time fault has occurred. Z must be rolled back to point A, Y must be rolled back to point B, and the right thread of X is aborted, as is shown in Figure 5.

There is another possibility, namely that S_1's commit guard does not include x_1, but it does include some other guard g_m from another process. Now we know that the event g_m is in the past of S_1, and that S_1 can commit only if g_m commits. But we do not yet know whether x_1 can commit. We will notify other processes that g_m precedes x_1 by sending a new type of control message called a *precedence message*. Eventually, one of three things will happen: (1) g_m will commit, in which case we can commit x_1, (2) g_m will abort, in which case we can abort x_1, or (3) we will receive a precedence message and learn that x_1 precedes g_m, in which case we will have detected a cycle and can abort x_1.

Figure 6 illustrates the first case. In this example, both process X and process Z have been optimistically parallelized. The return message to process X depends upon an earlier guess z_1 in process Z. Process X informs other processes which depend upon guess x_1 that the z_1 guess precedes the x_1 guess. Later z_1 commits, and then x_1 can commit also.

Figure 7 illustrates the last case. We begin the same way, but process X later learns from process Z that x_1 precedes z_1. Thus process X detects a cycle and aborts.

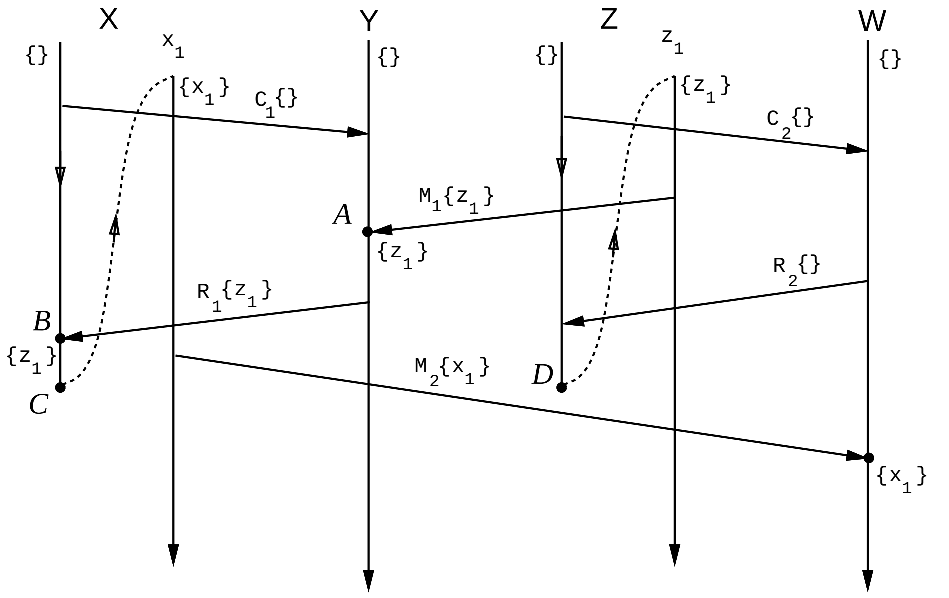

Figure 6: Successful Parallelization of Two Threads. One-way message sends (M_1 and M_2) are used as well as calls and returns. At point C the message PRECEDENCE(x_1,{z_1}) is sent to Z. At point D the message COMMIT(z_1) is sent to X and Y, which both commit their computations. Finally, X sends COMMIT(x_1) to W, which then commits its computation.

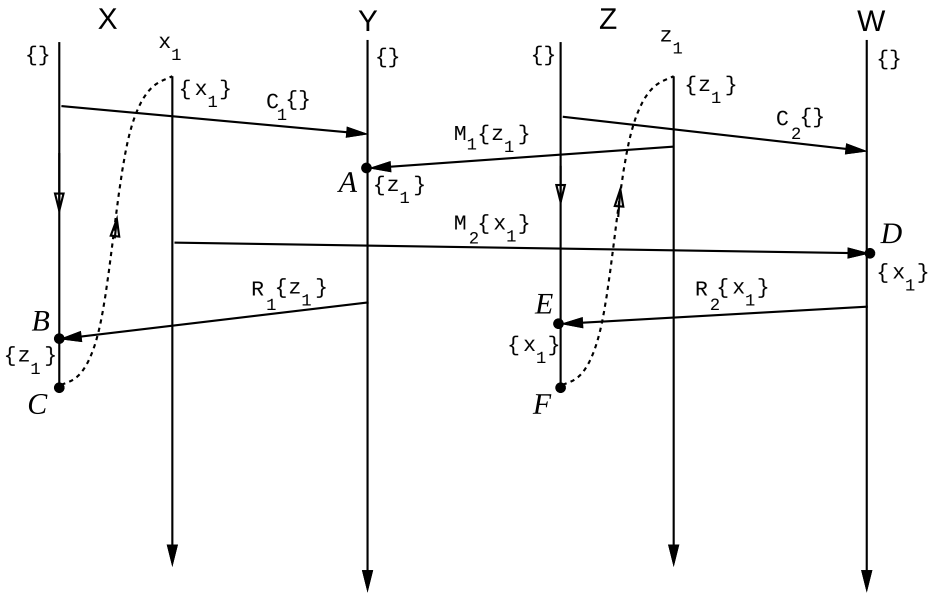

Figure 7: Aborted Parallelization of Two Threads. At point C $\{z_1\} \rightarrow \{x_1\}$ and at point F $\{x_1\} \rightarrow \{z_1\}$. The two left threads send PRECEDENCE messages to one another, and discover the cycle $x_1 \rightarrow z_1 \rightarrow x_1$. X sends ABORT(x_1) to Z and W, and Z sends ABORT(z_1) to X and Y. As a result, Y rolls back to point A, W rolls back to point D, and X and Z both abort their right-hand threads. The cycle includes M_2, R_2, M_1, R_1, and both dashed lines.

3.3 Liveness

In any protocol employing aborts, it is necessary to demonstrate that it is not possible to deadlock an otherwise live computation by rolling back infinitely often.

Our algorithm is live provided: (1) we assume a finite number of processes in the system; (2) we impose a limit L specifying the maximum number of times the same computation will be re-executed optimistically; when this limit is exceeded, that particular computation will be re-executed pessimistically; (3) commit, abort, and dependency messages are guaranteed to be propagated in finite time.

Theorem 1 *Subject to the above conditions, an optimistic parallelization of a distributed system will yield the same partial traces as the pessimistic computation.*

We first show that every individual commit guard predicate is committed or aborted in a finite period of time:

Every commit guard predicate x_i is associated with a fork. The left thread will either complete before the timeout period, or will time out. If it times out or completes with a value fault, x_i is aborted. If it completes without a value fault, then there may be a time fault. If there is a time fault, there exists a chain of processes $Y, Z, \ldots$ such that $x_i \rightarrow y_j \rightarrow z_k \cdots \rightarrow x_i$. Predicate x_i will be aborted after all left threads in the chain have terminated or timed out and messages have traveled a number of hops equal to the length of this chain. If any process appears more than once, say y_m and y_{m+k}, then either y_m will have already aborted before the fork y_{m+k}, or $y_m \rightarrow y_{m+k}$. If y_m has aborted, then x_i will eventually receive an abort message. If $y_m \rightarrow y_{m+k}$, then any dependency message for y_m received by process Y can be immediately propagated to the processes which depend upon y_{m+k}. Therefore the messages will propagate through a chain which never includes the same process more than once. Since there are finitely many processes, this chain must eventually terminate.

Similarly, if there is no time fault, then either some antecedent will abort, causing X to be aborted in finite time, or else commit messages will be propagated through a finite number of processes and will eventually reach process X.

We now show liveness of the transformation by induction on the number n of antecedent optimistic decisions prior to any particular optimistic decision of a computation:

If $n = 0$, then this optimistic decision must be a fork. It will either commit or abort in a finite time. If it has not committed after L repetitions, we simply refuse to fork, guaranteeing that there will be no abort.

Suppose that our transformation is live for $n \leq k$. Then if the $k + 1^{st}$ decision is a fork, the above reasoning applies. If it is the receipt of a message, then the message is from a partial computation with a maximum of k optimistic decisions. But we know that this partial computation is live, so we will produce the same results as the pessimistic algorithm by retrying the computation from the read up to L times, and then blocking the receive operation, accepting only committed messages. $\square$

4 Implementation

4.1 Data Structures

We use the following notation: Process identifiers are denoted by upper case letters, such as X and Y. The n^{th} fork of process x will be denoted x_n; the guess that gave rise to that fork will also be denoted x_n. The guess is that the corresponding left thread will complete successfully with no time fault and no value fault.

4.1.1 State Index

Each thread retains a *current state index*, a pair consisting of its relative thread number n, and an interval number $Interval$ which is incremented every time a message is received which introduces a new dependency. The process stores its largest assigned thread identifier as $MaxThread$.

4.1.2 Commit Guard Set

Each thread maintains a set $Guard$ of guesses that it depends upon. For the initial thread of a process, $Guard = \emptyset$.

The commit guard set contains all uncommitted guesses for all processes upon which the process depends. For instance, $\{x_3, x_5, y_9\}$.

A thread may depend upon many guesses by the same process, particularly if an optimization like call streaming is applied repeatedly. We would like to take advantage of the fact that a dependence upon x_5 implies a dependence upon all previous committed threads of x, and only store one guess per process in *Guard*. However, since x_3 may abort before or after x_5 has been forked, this would make it impossible to decide how to respond to the message ABORT(x_3).

The solution is to represent the guess identifiers as pairs, consisting of an *incarnation number* along with a thread identifier. The incarnation number is incremented every time the process aborts one of its own threads, and the thread index is reset to the index of the abort thread. Each guess is represented as a pair $x_{i,n}$ where i is the incarnation number and n the index of the thread. An *incarnation start table* records the index associated with the start of each incarnation. The incarnation start table can be used to determine the set of antecedents of a given event. For example, if incarnation 2 of process X begins at event 3, then the guess $x_{2,4}$ is known to be preceded by $x_{1,1}$, $x_{1,2}$, and $x_{2,3}$, but not by $x_{1,3}$. Receipt of $x_{2,3}$ can also be taken as an implicit abort of $x_{1,3}$. Because the incarnation start table records the abort history of each process, only the most recent guess from each process needs to be maintained in the commit guard set.

Incarnation numbers and their implementation are described in [9]; we will describe the algorithm without incarnation numbers for simplicity of explication, but they should be used in any real implementation.

4.1.3 Rollback Points

For each guess $g \in Guard$, we retain the state index at which the thread first became dependent upon that guess, denoted by *Rollbacks*[g]. If that guess is aborted, then we roll back to the end of the interval preceding *Rollbacks*[g], which is just before the receive operation that introduced the dependency upon g.

Note that if the rollback point is in a previous thread, then this (and all intervening threads) are aborted.

4.1.4 Commit Dependency Graph

For each thread we maintain a commit dependency graph, *CDG*, which is a DAG with the members of *Guard* as leaf nodes. When a thread terminates and knows the set of dependencies for the following thread, it broadcasts them and they are added by each receiver to its commit dependency graph.

For instance, if $Guard = \{x_1, y_7\}$ and the control message PRECEDENCE(x_1, $\{z_5\}$) arrives, then the edge $z_5 \rightarrow x_1$ is added to the *CDG*. If a cycle is detected in the *CDG*, then some guess must be aborted.

4.1.5 Commit Histories

Each process maintains commit information about each of the processes with which it communicates. For each guess of each of these processes, the history records whether it has *committed*, *aborted*, or is *unknown*.

Since most guesses are assumed to commit, this should be implemented as a sparse vector with missing elements assumed to be commits. We will treat it as a vector for simplicity. We write *History*[x_3] to denote the commit history entry for guess 3 of process x.

4.2 Events

4.2.1 Fork

A *fork* operation begins a new thread. A timer is set for the compiler-determined timeout interval. The state is copied into each new thread; then the compiler-determined predictor function is applied to determine the initial state of the right thread.

Interval is set to 0 and *MaxThread* is incremented by one and becomes the relative number n of the new thread x_n, so the new state index is $(n, 0)$.

The *History* and *CDG* are copied from the creating thread. The *Guard* is the union of the creating thread's *Guard* and the guess x_n. The *Rollbacks* vector is copied from the creating thread. *Rollbacks*[x_n] is assigned the value $(n, 0)$.

As an implementation issue, note that in the case of call streaming the fork can be performed *after* the call has been sent, rather than before, which will substantially decrease the likelihood of out-of-order message arrivals such as in Figure 4. Also, since the section of the process between the fork and join points is simply waiting for the return of the call, it is not necessary to make a copy of the state for the right-hand thread.

4.2.2 Send

When a message is sent, the *Guard* is appended to it. The entire *CDG* could be appended as well. This would allow earlier commit processing by the receiver at the expense of larger messages and more processing per message arrival.

4.2.3 Message Arrival

When a message arrives at a process, it is first checked that the message is not an orphan. The message is an orphan if

$$\exists_{g \in Guard} : History[g] = aborted.$$

Orphan messages are discarded.

From a correctness point of view, a message can be delivered to an arbitrary thread of the process, but we will often have information available which allows us to optimize the delivery decision.

If we have some knowledge that the message was intended for a particular thread, for example if this is the return of a call, we can check that the message does not depend upon some future thread. If it does, then we know that the future thread must be aborted, since a return which must be in its past has interacted with it.

Note that this is purely an optimization which allows early detection of aborts: if we do not check the message, then the caller thread will eventually time out waiting for the return and abort. For call streaming this optimization is straightforward since returns are easy to match up with their corresponding calls; in other cases this may be more difficult.

If the message passes these checks, then we deliver it to any thread which is waiting for a message of this type, provided that the message does not depend on the future of the thread. Generally, the message should probably be delivered to the *earliest* possible thread. For instance, if a message arrives for process a with its $Guard = \{x_5, y_3\}$, and the process is running its left thread plus three forks with guesses x_4, x_5, and x_6, then the message can only be delivered to threads x_5 and x_6, and preferably to x_5. Delivering it to x_4 would cause an abort because the end of x_4 precedes the beginning of x_5, and x_5 would therefore depend upon itself.

4.2.4 Receive

When a receive is executed one of the messages made available by the message arrival processing described above is received. If no message is available, the thread blocks.

When a message m is received by process X, it may introduce new dependencies. If

$$Newguards = Guard_m - Guard_X \neq \emptyset$$

then new dependencies have been introduced, meaning that a new interval has begun. *Interval* is incremented and for each $g \in Newguards$, g is added to the *Guard* and the *CDG*, and *Rollbacks*$[g]$ is set to $Index = (Thread, Interval)$, the new state index.

Note that if several messages are available, the one that introduces the fewest new dependencies (that is, the one for which $|Newguards|$ is smallest) should be chosen. This minimizes the chance that receiving the message will lead to an aborted state.

As an optimization, if we know that a message is a read message (does not cause the the process state to be modified), then we do not need to put it in *Rollbacks*; if aborted, we just remove it from the *CDG*.

4.2.5 Join

When a thread x_n of process X ends and there is a thread x_{n+k} running that logically follows it, the two threads are said to execute a *join*.

When a join occurs, guess x_{n+k} may be immediately committable, immediately abortable, or remain unknown.

First the verifier function is executed to check that the state at the end of thread n is consistent with the state that was used to begin thread $n+k$. If the verifier fails x_{n+k} is aborted: a control message ABORT(x_{n+k}) is sent, and thread $n+k$ and any dependent threads are terminated.

Otherwise, if $Guard_n = \emptyset$ then x_{n+k} does not depend upon any uncommitted guesses, and can therefore be committed. The control message COMMIT(x_{n+k}) is sent, thread n is terminated.

If $Guard_n \neq \emptyset$, then x_n depends upon guesses whose outcome is unknown. In this case, the message PRECEDENCE$(x_n, Guard)$ is sent, and thread n waits for other COMMIT, ABORT, and PRECEDENCE messages until it has enough information to decide to commit or abort x_n.

So far we have described these control messages as simply being "sent". They could either be sent by broadcast or by explicitly sending them to processes which are known to depend on the guard in question (this information could be recorded during message send processing). The former should work well in a local-area network where the threads are created relatively infrequently. The latter would be more appropriate in a wide-area network or when the number of threads created is large; the control messages could then be piggybacked on data messages being exchanged.

For simplicity of explication, we assume that control messages are broadcast. The broadcast itself need not be reliable, since the loss of a message will not cause incorrect commits. However, the broadcast must be live in the sense that if repeated broadcasts are made, a message will eventually be delivered.

4.2.6 Receive Commit

When the message COMMIT(x_n) is received, the history is updated so that $History[x_n] = commit$. x_n is removed from each *Guard* and *CDG*. Any predecessors of x_n are also removed from the *CDG* since they must also have committed.

Any threads that are waiting to complete a join are informed, and may commit as a result.

4.2.7 Receive Abort

When the message ABORT(x_n) is received, the history is updated so that $History[x_n] = abort$. Each thread is then checked in turn: if $x_n \in CDG$, it must be rolled back, and an abort message for that thread will be sent.

Let *Abortset* denote the members of *Guard* that follow x_n in the *CDG*. Then

$$a = \min\{r | g \in Abortset \wedge r = Rollbacks[g]\}$$

is the earliest state interval that depends upon x_n, and we roll back to that point, possibly aborting some guarded threads, including threads waiting to complete a join.

4.2.8 Receive Precedence

When the message PRECEDENCE$(x_n, Guard)$ is received, we set $History[x_n] = unknown$. The *CDG* of each thread is updated: for each $g \in Guard$, the edge $g \rightarrow x_n$ is added if either g or x_n is a node of the *CDG*.

If an edge added to the *CDG* creates a cycle, then a *time fault* has been detected. All threads in the cycle are aborted.

5 Related Work

Jefferson [4] first applied the concept of time-faults to a model in which the events of a system are totally ordered. Like Jefferson's Time Warp implementation, our approach is based on optimism and rollback. The two approaches differ in two ways.

Firstly, in a Time Warp distributed simulation, each individual process runs sequentially. There is nothing analogous to our guessing the future of a process and executing it in parallel with the present. However, there is no reason in principle why this could not be applied to Time Warp as well.

Most importantly however is that Time Warp imposes a single, totally ordered global virtual time throughout the system. Every event must be assigned both a send time and a receive time by the application. If two clients call a server then the server must process the calls in the assigned virtual time order or else the server will roll back. In our approach, there is only a partial order of events, which is determined during execution. We only roll back if our execution would result in a cycle in the happens-before relation.

In a distributed or loosely coupled parallel system, processes are compiled and instantiated independently. It is not feasible to impose a total ordering upon the computations, since this would require the sending process to decide upon a receive time for the message independent of the receiving process and any other senders. In contrast, the parallel simulation application for which Time Warp was originally designed can use the simulation time for events to provide the total order.

The ParaTran system [11] transparently parallelizes Scheme code using Time Warp. It replaces variables by stacks to support rollback, and automatically generates the timestamps required by Time Warp based upon the sequential dependencies of the program. As in Time Warp, there is a global total ordering of events as opposed to our partial ordering. The ParaTran methodology applies to a single sequential program, not a network of multiple sequential processes. In addition, there is no provision for increasing parallelism by using optimistically guessed values.

Concurrency control techniques, such as optimistic concurrency control[6], also guess that conflicts do not occur and abort computations when conflicts are detected. However, there are two differences between the correctness requirement for optimistic parallelization and the correctness requirement for concurrency control: (1) In concurrency control, all transactions must be serializable, whereas in optimistic parallelization, each individual process is separately serializable but the system as a whole is partially ordered; (2) In concurrency control, any serializable ordering is legal, whereas in optimistic parallelization only orderings which are consistent with $S_1; S_2$ are legal.

6 Conclusions

We have given an algorithm which is capable of transparently transforming a serial program $S_1; S_2$ into $S_1 \parallel S_2$. It is not necessary to prove statically that S_1 and S_2 do not interfere; our algorithm allows them to run in parallel and if a conflict is detected, aborts S_2 and re-runs it after S_1 terminates.

Because our transformations are transparent, we can achieve the performance benefits of complex procotols such as call streaming without requiring programmers to rewrite code when moving from a centralized to a distributed system. We allow programmers to use straightforward and familiar programming idioms, such as call statements, which are easy to reason about. Programming languages need not contain low-level primitives for explicitly managing concurrency.

The particular application of our algorithm to call streaming is particularly relevant to high-speed networks in which network latency will become an increasingly large fraction of the total time to complete a distributed computation.

In summary, our algorithm has the transparency properties of conventional parallelizing optimizations. However it is applicable to systems of distributed processes that are either not subject to static analysis or which are not considered parallelizable by static analysis.

References

[1] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer Usenix Conference* (July 1986).

[2] Allen, R., and Kennedy, K. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems 9*, 4 (October 1987).

[3] Hoare, C. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, N.J., 1985.

[4] Jefferson, D. R. Virtual time. *Transactions on Programming Languages and Systems 7*, 3 (July 1985), 404–425.

[5] Kuck, D., Kuhn, R., Padua, D., Leasure, B., and Wolfe, M. Dependence graphs and compiler optimization. In *Proceedings of the 8th ACM Conference on Principles of Programming Languages* (January 1981), ACM.

[6] Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[7] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[8] Strom, R. E., Bacon, D. F., Goldberg, A., Lowry, A., Yellin, D., and Yemini, S. A. *Hermes: A Language for Distributed Computing.* Prentice Hall, Inc., February 1991.

[9] Strom, R. E., and Yemini, S. A. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems 3*, 3 (August 1985), 204–226.

[10] Strom, R. E., and Yemini, S. A. Synthesizing distributed and parallel programs through optimistic transformations. In *Current Advances in Distributed Computing and Communications*, Y. Yemini, Ed. Computer Science Press, Rockville, MD, 1987, pp. 234–256.

[11] Tinker, P., and Katz, M. Parallel execution of sequential Scheme with ParaTran. In *Lisp and Functional Programming Conference* (1988), pp. 28–39.

[12] United States Department of Defense. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815-1983 ed., February 1983.