Worlds in Collision: A Mostly Functional Model of Concurrency Control and Recovery

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436

August, 1990
This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

Many techniques of concurrency control and crash recovery are implementation instances of an abstract "world tree" model. This abstract model factors out most low-level implementation details of transactions, intentions, shadowing and logging, so that the essential issues like conflict resolution can be seen more easily. Parallel synchronization can often be seen as a merging of disparate worlds after resolving their differences. Nested transactions fit easily into this model, as does crash recovery. A "lazy shallow binding" implementation model of world trees is described which offers good performance during both normal and abnormal execution modes.

1. INTRODUCTION

We present a "multiple worlds" model of a parallel/distributed processing system "universe", which abstracts out the implementation details of concurrency control and crash recovery, so that one can more easily focus on the more important issue of conflict resolution. We show how crash recovery is an instance of synchronization in this model, and show how a "lazy shallow binding" implementation can provide good performance both during normal execution, as well as during crash recovery.

We start with a very general model of states and actions, and then demonstrate the application of these ideas to a database shared among several processes.

2. MOSTLY FUNCTIONAL HARDWARE DESIGN

Digital hardware designers, as a group, are one of the greatest repositories of knowledge of how to design parallel systems. These designers usually have significant constraints in the areas of cost and parts availability, but they also have a great deal of freedom in the area of concurrency.

Asynchronous logic has been touted for over 25 years, because it has the potential to maximize the concurrency available in certain computations.. Yet hardware designers have firmly rejected asynchronous logic. Because similar mechanisms have been advocated for some concurrent programming languages, it should be a significant priority for parallel software designers to understand the reasons behind the lack of acceptance of asynchronous logic.

One of the goals in hardware design is the minimize the amount of synchronization that is required—especially ambiguous synchronization, where a choice must be made among alternatives. Not only is the synchronization logic itself expensive, but ambiguous synchronization can produce system failure through "runt pulses". If a substantial fraction of the system is clocked in a synchronous manner, then within this synchronous domain no synchronization logic is necessary (other than the clock itself), and no "runt pulses" can be generated. Any interaction with the outside world can cause failure, however; these interactions can be minimized and localized within a few well-confined modules.

Asynchronous logic can offer greater speed for determinate computations, because every circuit operates at its actual rate—not at a speed determined by its slowest element. In asynchronous logic, however, many straight-forward tasks become choices requiring ambiguous synchronization, which leads to a greater likelihood of "runt pulse" failure. As a result, the possibility of failure is pervasive, rather than being localized where it can be more easily dealt with.[1]

In addition to the possibility of failure, asynchronous logic is difficult to work with because *state* is distributed throughout the system. This makes it difficult to recover from failure, and nearly impossible to test or debug, because much of this state is *inaccessible*. Since testability is essentially to modern chip-building, it is required that any state be capable of being sensed and manipulated.

Asynchronous logic is difficult to debug, because the switchings of its logic elements do not form regular patterns. This lack of patterning makes oscilloscopes and logic analyzers useless. While new tools for debugging asynchronous logic may make the job easier, such tools seem to require more sophistication—and hence more cost—than synchronous debugging tools.

If asynchronous logic is so difficult to deal with, then what techniques *do* digital designers find useful? High performance digital systems operate in well-defined cycles, with heavy use of *pipelining*. In a pipelined system,

[1]This localization is actually a myth, because runt pulses can propagate throughout the system. Within certain modules, however, the probability of runt pulses can be minimized using more expensive circuitry.

© 1990, 1993 Nimble Computer Corporation

long stretches of combinational logic (logic without state) are punctuated by clocked registers, which *reclock* the computation. These pipeline registers solve two problems: they allow multiple computations to proceed at the same time, and they reclock (i.e., synchronize) the pulses so that they travel together as a group. A pipelined system can be compared to a boulevard with synchronized traffic lights, where each group of cars is separated from the preceding and following groups by the traffic lights. Cars which travel too fast will find that they are required to wait at every light for their cohorts. A well-designed synchronous digital system has no counterpart to slow cars.

Even in sequential circuitry, state is not distributed. For example, modern digital designers use a combination of a register and a ROM or a register and a PLA to design a state machine. Since ROM's and PLA's are stateless, the state of the state machine is localized in the register.

What lessons can the software system designer learn from the hardware engineer? First, any *state* in a parallel system should be highly accessible and well organized. Constructs which introduce implicit state into a program should be considered with great suspicion, and methods should be available to set and reset any implicit state. Second, state should also be used sparingly on a dynamic basis, with a reasonable amount of computation occuring between constructs utilizing state. Thirdly, most synchronization should be implicit—i.e., not require the full expense of arbitrary ambiguous synchronization—thus allowing ambiguous synchronization to be restricted to a few well-defined modules. Fourthly, reclocking can be advantageous, because while it slows some of the computations down, it makes the patterns of the overall system more predictable and controllable.

We therefore advocate a style of programming called "mostly functional programming". Unlike purely functional programming, which attempts to completely finesse the issue of state, we realize that state is an essential part of the "real world" and must be dealt with. For example, synchronization of two processes requires at least one bit of state—which process arrived first—and so state cannot be avoided in a parallel system which performs any communication. Since state is a "necessary evil", we seek to control it by making it accessible and utilize state-accessing and state-changing operations as sparingly as possible. By organizing states into "worlds", and by performing as much computation as possible using purely functional techniques, we believe that we can come closer to the desired goals of efficiency, reliability, predictability and testability.

3. THE MULTIPLE WORLDS MODEL OF A PARALLEL SYSTEM

Some have advocated the use of "pure" (side-effect-free) programming to take advantage of parallel processing capabilities. While purely functional languages can offer substantial advantages for many computations, it is unreasonable to attempt to perform *every* computation in a side-effect-free manner, just as it is unreasonable to design a digital system using only loop-free combinational logic. In particular, computations which interact with the "real world" must deal with its state, even if the computation itself has no state. For example, even a purely functional programming language must perform input and output operations, and these operations involve side-effects. These side-effects can sometimes be finessed by considering the output as a *stream* of results transformed from a *stream* of inputs. The stream model captures only a very small portion of real computer applications, however. A robot, for example, *senses* the world's *state*, and performs *actions* which are intended to change the world's *state*. There is therefore no way of finessing the problem of the world's state in a robotic application.

A real-time system must therefore deal with the state of the "real world". If such a real-time system must also be *persistent*, this adds another world that the system must be aware of—the state of the world at the last *checkpoint*, *commit*, or *log*. Once we are dealing with at least two different worlds, we might as well deal with an arbitrary number.

Our *worlds* are the carriers of side-effects. Every side-effect is performed in a specific world, whether implicitly or explicitly specified. Every side-effect is an *action*, which "modifies" the state of the specified world in a well-defined way. An action does not actually modify the world itself, but creates a new world in which the action has already been performed, and then side-effects a "world pointer" to point to the new world.[2]

For (almost) every action, we have an equal and opposite *reaction*, which "undoes" the action; we also call reactions *antidotes*. If an action has an antidote, then it is called *reversible*; otherwise, it is called *irreversible*.[3] Most real-world actions have antidotes, but they might not be very palatable. For example, if an automatic teller machine gives out an extra $20 by mistake, then the antidote is to chalk up the loss to the "ATM mistakes" account; the accounts remain in balance. If the ATM gives out $5,000 by mistake, the antidote might involve politely asking for the money back before chalking up the loss. Of course, a simple imperative assignment has a simple antidote—restoring the previous value of the assignable cell. Whenever possible, antidotes should be computed automatically by a compiler in order to avoid inconsistencies.

The existence of antidotes for actions allows for much greater flexibility in moving from one world to another—i.e., *navigating*. Antidotes also allow for the greater use of *optimistic* rather than *pessimistic* protocols,

---

[2]As a result, the only "real" side-effects are those to these world pointers.

[3]An obvious extension would assign costs to actions and antidotes, so that navigation costs can be minimized.

© 1990, 1993 Nimble Computer Corporation

with their greater potential for parallelism, because synchronization failures need not be catastrophic. Since navigating among internal "imaginary" worlds is usually cheaper than performing external actions and antidotes,[4] the robot can envision multiple worlds on the way towards creating a plan for moving from the current real world to a real world in which some goal is satisfied. Once an (imaginary) world has been conceived together with a sequence of actions, then a *commitment* can be attempted, which tries to actualize the imaginary world by executing the actions. The commitment process executes every step in the sequence atomically, so that if the imaginary world cannot be actualized, then the previous world can be recovered by atomically executing the antidotes.

Multiple internal worlds are creating by *branching* or *splitting* an existing world. Once multiple branches have been created, they cannot communicate in any normal way, because communication and/or synchronization *must* involve side-effects, and we have restricted side-effects to a single world. All communication/synchronization among worlds is achieved through the *merging* of worlds. Merging worlds involves resolving the differences between the worlds so that they can be considered the same world—i.e., unified. If these conflicts cannot be resolved, then one of the worlds dies.[5] Because merging is achieved by making it appear as if the worlds never diverged, the *universe* is always a *tree* of worlds in which the branches are sequences of actions/antidotes that allow navigation among the worlds.

Persistence is achieved by periodically merging a world with persistent memory, which is equivalent to *checkpointing*. Since the persistent memory world is completely passive, it is always an ancestor of a current world. Therefore there cannot be any conflict between it and a current world, so the checkpoint merge is always trivial and succeeds unless the system crashes during the merge. Notice that this view of persistence is completely silent about what is logged and when it is logged—whether intended values or previous values. This is because the existence of antidotes makes it possible to operate correctly in either mode, or even switch between the modes at will. The section "External States and Actions" deals with this issue.

We note that different worlds may or may not differ in the "real time" assigned to them, and there could very well be multiple conflicting worlds with the same "real time". Of course, it may be difficult to reconcile two of these worlds, especially if one of them is the "real world".

We also note that when merging conflicting worlds, the "real world" does not necessarily win! This is because it may sometimes be cheaper to "roll back" the real world to a known state, than to recompute some internal plan world. For example, it is often easier to "re-initialize" a balky piece of equipment than to diagnose its current state.

We have discussed the concept of *world* rather than the more traditional *process*, because the only visible manifestation of a process is the state of its world—i.e., the current state of its mutable objects. We don't particularly care about any state of a process that is not stored in a world—e.g., a stack pointer or register contents—because this state will be lost in a crash anyway. In other words, we do not bother with the notion of *transient* state, because it is not a well-defined concept in our failure-ridden universe.

One can also conceive of multiple "real worlds" in which different portions of the "real world" can be dealt with independently, in parallel. Such a model opens up the possibility of inconsistencies which are visible to the "real real world", however. For example, different terminals in an ATM network may be modelled by different "real world" processes. Two people who communicate via radio both attempt to remove all of the money from the same bank account at the same time. If the database uses optimistic synchronization, then both people will get their money, in which case one of the ATM's will abort and have to ask for the money back! This situation can be avoided by keeping only one "real world", in which case one of the ATM's will hang until the other has finished. However, the "one world" assumption also creates a massive serial bottleneck in front of the database. This serial bottleneck can be removed by a judicious application of detailed knowledge of the database—e.g., different bank accounts are independent—but in general, any redundancies in the database can be exploited to exhibit transitory inconsistencies.[6]

The simulation of a physical system can avoid transitory inconsistencies by appealing to the cornerstone of general relativity—causality cannot travel faster than the speed of light—to make sure that inconsistent states cannot be observed within the system. While modern distributed computer systems do utilize light and microwaves to communicate and synchronize, the effective speed of these modalities is much lower than the speed of light, and this speed difference can be exploited by a sophisticated larcenous observer to notice inconsistencies in the distributed system. Such an observer would require excellent knowledge of the internal workings of the system, and access to excellent communications, but given the possibilities for profit, these resources are available. For example, it has been rumored that spy satellite information has been used to "beat" the commodities market computers. Thus, any

---

[4]Otherwise, what's a simulation for?

[5]Usually by committing suicide, but there are instances where a world must be murdered. (Sorry about the violence, but that is the nature of side-effects.)

[6]Most distributed processing protocols rely on approximations to global clocks [Lamport78] to avoid falling victim to fast observers with larcenous intent.

© 1990, 1993 Nimble Computer Corporation

relaxation of the "one real world" assumption should be accompanied by a mathematical proof that the attendant protocols cannot be compromised by these larcenous observers.

Our model handles optimistic transactions quite naturally. In the simplest case, two worlds split and each performs some computation. At some point later, an attempt is made to merge the worlds, and if there is any conflict between the worlds, one of them is aborted, which causes it to "roll back" to the point where it split from the winning world. Because we do not consider the reading of an assignable variable to be a side-effect, we define conflict between worlds as any visible differences. We can capture the more traditional notion of transaction conflict if we treat each process as building (through assignments) its own private cache of the shared world. Then, the merging process would notice any conflicts between a read in one world and a write in the other through their conflicting actions on their private caches, and cause one of the worlds to abort.

The way we have described the world merging process makes it appear as if it is quite expensive. It need not be, however. Each world could be assigned a unique "time stamp" at the time it is created through splitting, and any merge can always be resolved by killing the world with the highest time stamp. This policy gives precedence to the earliest world to begin a transaction. Another cheap policy would be one which always resolves the merge in favor of the requestor of the merge. This policy gives precedence to the world which commits the transaction first.

Long-lived transactions have been identified as a problem in CAD environments, where multiple designers "check out" designs for long periods of time—e.g., weeks—and there is a probability of nearly one that when the design is "checked in" that some version of it will be aborted, destroying a great deal of effort in the process. We see no general algorithmic solution to this problem, because any cheaper resolution of the conflict requires deep knowledge of the particular CAD domain. For example, if two designers attempt to place two parts in the same physical location, one of the designs must yield, and this is normally accomplished through negotiations at Monday morning engineering meetings—not through a concurrency control algorithm. One can reduce the amount of wasted effort through more frequent synchronizations, but since the synchronizations themselves take a substantial amount of effort, there is a trade-off between wasted effort and excess synchronization. The system designer should have control over the amount of synchronization so that the amount of useful work can be maximized.

4. A MULTIPLE WORLDS MODEL OF A SHARED DATABASE

We achieve accessibility and controllability of all internal state in the form of assignable cells by incorporating them into a single "database", which is shared by a number of parallel processes. Because one process can read what another process has stored into this database, it becomes a channel of communication among the processes.[7] As a result, we conform to the "blackboard model", which has been profitably used for experiments in parallel artificial intelligence.[8]

Due to the requirements for *consistency* of the database, any database updating visible to other processes must be performed in an organized manner. In our model, any updating requires the explicit *agreement* or implicit *permission* of all of the other processes. If this agreement/permission cannot be achieved, then the updating process must back down, roll back, or otherwise fail.

We perform database updating through the *commitment* of a *transaction*, which either updates the database if it succeeds, or returns failure without updating the database. A failure signal is interpreted by the updating process as a signal to abort, roll back or restart the process with new assumptions.

Our model should be considered *optimistic*, since we do not require that a process acquire exclusive access to the database in order to perform updates. *Pessimistic* concurrency control can easily be simulated, however, for transactions that would be prohibitively expensive or impossible to roll back. Using optimistic concurrency control, the acquiring process simply updates an "exclusive access" flag in the database to its own process identifier; if the commitment is successful—i.e., the transaction is not aborted, then the acquiring process has achieved exclusive access to the database. All other processes will eventually fail when they attempt to update the database and find it locked.

Each process starts with the same global world view of the database, but the more it computes, the further its world diverges from the initial world. Since other processes also started with the same global world view, but have diverged, we develop a *tree* of different worlds, where the root of this tree is the initial global world view of the database. We can define a notion of *distance* between two worlds as the sum of the distances from the worlds to their closest common ancestor; we later discuss the metric for this distance. From time to time, a process may wish to *synchronize* with another process. This synchronization requires active negotiation between the processes to agree

[7] A programming language may provide additional constraints on variable visibility; we do not attempt to model these constraints here.

[8] Our model does not require a closely coupled, shared-address-space machine, but we have found it simpler to explain it in these terms.

© 1990, 1993 Nimble Computer Corporation

on a uniform view of the database. If synchronization is achieved, then both processes proceed with the new uniform world view. If synchronization is not achieved, then one of the processes must die.

We utilize a *monotonicity* assumption: if two processes which are close to each other in the tree cannot synchronize, then there is no hope that they can synchronize with processes further away in the tree. Similarly, if two processes close to each other can synchronize, then the basis of their agreement is also acceptable to other processes close to them in the tree. As a result of our monotonicity assumption, we can preserve the tree-like character of our universe of worlds. Whenever two processes synchronize, they change their histories to make it appear as if they always agreed, and in so doing, they bring any offspring processes closer together as a side-effect of their own agreement.[9]

This monotonicity is an extremely strong requirement, which cannot easily be achieved. As a result, most attempts at synchronization will result in the destruction of one of the processes. The choice of which process succeeds and which process fails can be made in many ways. One can assign to each process a distinct, fixed priority, which resolves all disputes among them. A fairer method is a FIFO policy, where the first process to reach the synchronization point will succeed if agreement cannot be reached. If several processes are incrementing a global variable with their own increments, then synchronization may involve agreeing to sum all of the increments when updating the global variable.

Explicit agreement is very expensive, because the agreement negotiations require the active involvement of the synchronizing parties. In some cases, however, it is possible for a process to give implicit *permission* in advance. Suppose that we have two processes A and B updating the shared database, and suppose that they currently agree on the state of the database—i.e., they share the same global world view. To begin a transaction, both processes store the global world view identifier and then proceed to update their own views. At some point in the future, one of the processes will attempt to commit his world view—i.e., synchronize his world to the rest of the processes by requiring them to accept his world. Commitment is only possible, however, if his world is consistent with the committed database—i.e., if the global committed world is the same as the world view he stored when he started his transaction. If the world view identifier stored in the global world view variable is not the same as his initial world, then another process has already updated the database, and he must abort. On the other hand, if the global world view is the same as his initial world view, then he can change the global world view to be his local world view, and any other transactions which attempt to update the global world view will eventually abort when they find the global world view is no longer consistent with their local worlds. In other words, all processes have blanket permission to update the global world view to a new world view which is consistent with the global world view, without having to explicitly synchronize with all of the other processes.

Our database "universe" thus holds a number of possible "worlds", one for each process, and these worlds form a tree as a result of their diverging histories from a single shared world. We can move from one world to another by updating the database or rolling back; these are the actions that can change the database. One may have a distinguished process which is the "real world", which has additional, hidden state. Changes to the "real world" correspond to I/O activity, and it is desirable that these changes have antidotes.

5. THE CONCRETE DATABASE MODEL

Our concrete model for a database consists of a file of fixed-length pages; i.e., a vector D of N *pages*. A program performs *read* and *write* operations, which randomly read or write a single page of the file when given the index of the page. We further assume that a read or write operation takes O(1) (i.e., constant) time and is *atomic*, in the sense that the read operation returns with the entire page, if it returns, and a write operation writes an entire page, or writes nothing if it fails.

- read(i), $0 \le i < N$, produces the i'th page of the database;
- write(i,p), $0 \le i < N$, p a page, updates the contents of the i'th page of the database with p.

We consider the database to reside in persistent (usually secondary) storage, which means that it is immune to hardware and software crashes. Methods for achieving this persistency abstraction have already been presented in the literature; we will focus entirely on consistency issues.

In addition to the database itself, we will have occasion to persistently store certain other objects, including dynamically allocated *nodes*, which may point to other dynamically allocated persistent nodes, and *root* pointers, which are variables whose contents are persistent. We assume that nodes of fixed size can be atomically allocated and initialized in O(1) time, that pointers can be dereferenced in O(1) time, and that root pointers can be atomically accessed and atomically assigned in O(1) time. We do not discuss the details of allocation, nor do we discuss the freeing of dynamically allocated nodes.

Our database is organized into multiple *worlds*, which each denote a "version" of the database. A world can be queried using *lookup*, and updated using *nupdate*. A world pointer can be changed using *assign_world*.

- lookup(i,W), $0 \le i < N$, produces the i'th page of the database world W;

[9] Such synchronization may later prove fatal to some offspring, even though the parent is spared.

© 1990, 1993 Nimble Computer Corporation

• update(i,p,W), $0 \leq i < N$, p a page, updates the i'th page of the world W with p to produce a new world W'; lookup(i,W) remains constant;

• assign_world(W,W') is atomically W:=W';

• nupdate(i,p,W), is atomically assign_world(W,update(i,p,W)).

Below is the Ada specification [Ada83] for our database.[10]

```
package database is
 type page is private;
 subtype index is 0..(N-1);
 type world is private;
 null_world: constant world;
 function lookup(i: index; w: world) return page;
 function update(i: index; p: page; w: world) return world;
 procedure assign_world(w: out world; w: world);
 procedure nupdate(i: index; p: page; w: in out world);
 end database;
```

We will discuss three different implementations for implementing worlds, which each have different timing characteristics.

Multiple Copies (Acquaintance Vectors).

The most straight-forward implementation of worlds involves making a separate copy of the entire database for each world; i.e., each world is implemented as a separate vector. Switching from one world to another takes O(1) time, as only a pointer to the appropriate world need be changed. Looking up a page of a world also takes O(1) time, since this operation is equivalent to a primitive read operation. Creating a new world (updating) requires O(N) time, however, since the entire database must be copied.

Deep Binding.

The deep binding implementation of a multi-world database uses a "change list" which encodes changes made to the initial world; and the nodes of this "change list" are stored in persistent storage along with the intial world. As each update is processed, a *delta* node is allocated which has the page number, the page value, and a pointer to the previous world. Lookup in this model requires that the change list be searched, and the first delta node whose page number matches provides the page value. If no such delta node is found, then the value of the page in the initial world is used.

Switching from one world to another in this model takes O(1) time, since only a pointer to the appropriate delta node must be changed. Looking up a page of a world can take O(M) time, where M is the number of steps in the program. This is because a program of M steps can create O(M) delta nodes, which must be sequentially searched. Creating a new world (updating) requires O(1) time, since a fixed-size node must be allocated, where this size is independent of both the size of the database and the length of the program. Below is the Ada code for the deep binding version:

```
package body database is
 -- Deep Binding Implementation.
 procedure assign_world(w: out world; nw: world) is
  begin w := nw; end;
 function lookup(i: index; w: world) return page is
  begin if w.parent=null then return read(i);
        elsif w.binding.index=i then return w.binding.page;
        else return lookup(i,w.parent); end if;
  end lookup;
 function update(i: index; p: page; w: world) return world is
  begin return new delta_node'(new binding_node'(i,p),w); end;
 procedure nupdate(i: index; p: page; w: in out world) is
  begin w := update(i,p,w); end;
 end database;
```

Shallow Binding.

The shallow binding implementation of a multi-world database also uses a "change list", but this list encodes the *previous* values, rather than the current values, which are kept up-to-date in the single copy of the database. As each update is processed, an empty delta node is allocated for the new world, while the page number and old page value and a pointer to the new world are installed in the delta node for the old world. Lookup in this model will find

[10] We use Ada because it is well-defined, widely used, and appropriate parallel constructs.

that the current value is always in the current database, while switching worlds requires tracing the change list and swapping values in the database with those in the change list.

Switching from one world to another in the shallow binding model takes O(M) time, since the change list to be traversed can be O(M) long. Looking up a page in the current world takes O(1) time, since it is readily accessible in the current database. Creating a new world (updating) requires O(1) time, since a fixed-size node must be allocated, where this size is independent of both the size of the database and the length of the program. Below is the code for the shallow binding version.

```
package body database is
 -- Shallow Binding Implementation; see [Baker78c].
 function onestep(nw,ow: world) return world is
  opage: page;
  begin
   nw.parent:=null; ow.parent:=nw; swap_binding(nw.binding,ow.binding);
   opage:=read(ow.binding.index); write(ow.binding.index,ow.binding.page);
   ow.binding.page:=opage;
   return nw;
  end onestep;
 function reroot(w: world) return world is
  begin if w.parent=null then return w;
        else return onestep(w,reroot(w.parent)) end if;
  end reroot;
 procedure assign_world(w: out world; nw: world) is
  begin w := reroot(nw); end;
 function lookup(i: index; w: world) return page is
  begin
   assert(w.parent=null); -- Can only be called in shallow context.
   return read(i);
  end lookup;
 function update(i: index; p: page; w: world) return world is
  begin
   assert(w.parent=null); -- Can only be called in shallow context.
   return onestep(new delta_node'(new binding_node'(i,p),w),w);
  end update;
 procedure nupdate(i: index; p: page; w: in out world) is
  begin w := update(i,p,w); end;
 end database;
```

It is essential that `onestep` be performed in a fashion which is atomic both to concurrent processes as well as to crash recovery.

Crash Recovery.

A program wishing to be protected against crashes must keep track of (at least) two different world pointers—its *current* world pointer, which indicates the world the program is currently updating, and the *committed* world pointer, which indicates the latest *committed* world of the database. The committed world pointer is stored in the persistent database.

The program starts with both the current world and the committed world pointers pointing to an initial database world. We assume that no database activity takes place outside of a *transaction*, which is a delimited section of the program. At the beginning of every transaction, then, both pointers point to the same world of the database. During the transaction, however, multiple lookups and updates occur to the current world, thus producing a succession of new worlds. The transaction is committed by assigning the current world pointer to the committed world pointer.

Crash recovery is extremely simple in this model: the committed world becomes the current world using *assign_world*. No more is necessary, because each stored world is always consistent.

It is instructive to examine the workings of each of the three implementation models during crash recovery. In the multiple copies implementation, the transaction begins by setting the current pointer equal to the committed pointer. During each update, copies of the entire database are made; copying is an atomic (although arbitrarily long) operation. Finally, crash recovery involves resetting the current pointer to the committed pointer, and hence is O(1) While the multiple copies implementation will work, it is quite slow and requires O(M•N) persistent storage.

The deep binding implementation does not require a complete copying operation for any part of our scenario. The initialization of the current pointer is O(1), as are all of the update operations. The amount of storage utilized is

O(M), and the time cost of looking up a page can be O(M). Crash recovery is O(1), as a world change in this implementation is simply a pointer change.

The shallow binding implementation does not require a complete copying operation, either. The initialization of the current pointer is O(1), since we assume that at program initiation, the current world of the database is already the same as the committed world. Lookups and updates during the transaction are each O(1), and the program requires O(M) persistent storage. Crash recovery, however, requires a context switch from the current world back to the committed world. Since this involves a linear scan of the delta nodes between the committed world and the current world (the pointers go from old to new in shallow binding), it requires O(M) time to recover from a crash. While normal operations of a program in the shallow binding model are quite fast, crash recovery can be slow. Crashes usually have a low probability, though, so the shallow binding model has the lowest expected cost.

Shallow binding has a problem, however. If the system were to crash during crash recovery, it is not clear what could be done to pick up the pieces. We can solve this problem, as well as the problem of quickly recovering from a crash, through the mechanism of "lazy shallow binding".

Lazy Shallow Binding

We will henceforth ignore the multiple copies implementation, and concentrate on a synthesis of the deep and shallow binding models. This "lazy shallow binding" model will use a lookup mechanism similar to that of deep binding; i.e., a changes list is searched to find a value, and if no value is found in the changes list, then the value is read from the current database. As a side-effect, however, the world where the value was found is made the root of the world tree through a process called "re-rooting". Re-rooting rearranges the database for more efficient lookup, but does not change the value of any page in any world. The update mechanism is the same as the shallow binding implementation—namely, that a delta node is created which hold the old value, and the old world points to the new one. We will utilize the context switch offered by the deep binding model—i.e., a simple pointer change to implement context changes. Below is the code for this model:

```
package body database is
 -- Lazy Shallow Binding Implementation.
 procedure assign_world(w: out world; nw: world);           -- like deep.
 function onestep(nw,ow: world) return world;               -- like shallow.
 function reroot(w: world) return world;                    -- like shallow.
 function lookup(i: index; w: world) return page is
  begin if w.parent=null then return data(i);
        elsif w.binding.index=i then return lookup(i,reroot(w));
        else return lookup(i,w.parent); end if;
  end lookup;
 function update(i: index; p: page; w: world) return world is
  begin return reroot(new delta_node'(new binding_node'(i,p),w)); end;
 procedure nupdate(i: index; p: page; w: in out world);     -- like shallow.
  end database;
```

Our transaction now operates similarly to the shallow binding model above, except that every lookup will check a change list, and always find it empty. Crash recovery is now instantaneous, but the first few lookups during the running of the next program will be slower, because a side-effect of the lookup is a rearrangement of the world tree. The re-arrangement of the tree is performed in a series of atomic O(1) steps, and in between each of these steps the world tree is consistent, so additional crashes during a restart cannot hurt the database.

A programmer may wish to re-root more or less aggressively than deep, shallow or lazy shallow policies offer. Since re-rooting does not change the program-visible behavior of the database, giving the programmer control over this operation cannot cause any problems.

External States, Actions and Antidotes

If we include external states and actions in our worlds, then we can see a dramatic difference in the implementation models. In the shallow binding model, external actions are taken immediately and antidotes are logged in the "changes list", while in the deep binding model, external actions are logged in the "changes list" and are "envisioned", but never performed!. In other words, shallow binding is optimistic with respect to external actions, while deep binding is eternally pessimistic. By using a deep binding model in conjunction with user-controlled re-rooting, or by using a lazy re-rooting model, the programmer can reduce the expense of rolling back external actions when a transaction aborts.

Optimistic Parallel Synchronization

We now consider the problem of multiple parallel threads of computation accessing the same database. Each parallel process accesses the shared database within the confines of an atomic *transaction*, which guarantees that either the whole transaction succeeds (or *commits*) or the whole transaction fails (or *aborts*). The purposes of a transaction are to keep any intermediate database states within the transaction from becoming visible to other parallel processors through their normal activity of reading and updating the shared database, and to make sure that no other process interferes with the operation of the transaction.

A traditional model of correctness is that of *serializability*. We presume that if a single process runs by itself, then whatever changes it makes to the database by itself are correct. If multiple parallel transactions are attempted, then serializability states that we want the result of the whole computation to be the same as if some serial order were chosen for all of the transactions, and that only one transaction be executing at any one time. In other words, serializability is equivalent to a mutual exclusion model for transactions, where each transaction can assume to have the database to itself.

The traditional mutual exclusion method for transactions is usually achieved through *locking*, which locks out other processes from any access to the database during the execution of one transaction. Locking can be performed on less than the entire database, but sophisticated algorithms are required to avoid deadlock. Locking is called *conservative*, because it is unwilling to tolerate the possibility of conflict.

An alternative to a conservative transaction protocol is optimistic execution, in which the parallel processes all forge ahead, accessing the database as they go. When a process reaches the end of a transaction, it decides to either *commit* or *abort*. If its computation does not conflict with other transactions, then it will commit, while if it determines that there has been a conflict, then it aborts.

Below is an Ada skeleton for optimistic parallel synchronization using our multiple worlds:

```
task transaction is
 entry start_xaction(begin_world: out world);
 entry commit_xaction(end_world,begin_world: world; aborted: out boolean);
 end transaction;
task body transaction is
 current_world: world := initial_world;
 begin
  loop
   select
    accept start_xaction(begin_world: out world) do
     begin_world := current_world;
   or
    accept commit_xaction(end_world,begin_world: world; aborted: out boolean)
     -- Here, conflict is resolved in favor of process which commits first.
     do if begin_world = current_world
        then current_world := end_world; aborted := false;
        else aborted := true; end if;
   end select;
  end loop;
 end transaction;
task parallel_process is end;
task body parallel_process is
 begin_world,my_world: world := error_world;
 aborted: boolean;
 begin
  loop
   start_xaction(begin_world);           -- Get the current committed world.
   assign_world(my_world,begin_world);   -- Install it as our world.
   << Process a transaction involving lookups and updates of the form:
      nupdate(i,p,my_world);             -- Update my_world. >>
   commit_xaction(my_world,begin_world,aborted);
   if not aborted then << atomically move to next transaction.>> ; end if;
  end loop;
 end parallel_process;
```

Our tree-shaped universe of worlds easily allows for a number of processes and the potential of nested transactions, unlike systems based on linear intention or commit lists. Our model also makes no commitment about which of the binding systems is used to implement the database.

6. CONCLUSIONS

We have described a "tree of worlds" model of a parallel system which brings out the important issue in concurrency control—conflict resolution. Nested transactions [Reed78] are a trivial consequence of the *tree* of worlds. We have shown how crash recovery is a kind of synchronization, and how many implementations of crash recovery are variations on one of three traditional variable-binding implementation models. A trivial extention to the model allows for the handling of side-effects more general than assignment. By providing more abstraction in the analysis of parallel systems, we hope to orthogonalize several issues and thereby make the systems more modular and more understandable.

7. COMPARISON WITH PREVIOUS WORK

Lazy shallow binding is a straight-forward application of the techniques of [Baker78c], which investigated similar issues in the management of multiple variable-binding environments. The *re-rooting* scheme, introduced in that paper, is identical to the one shown here. [Baker90b] shows another application of re-rooting for the efficient implementation of arrays in a purely functional language.

Hanson and Lamping [Hanson84] introduced the notions of *dynamic wind* and *state space* (also called *dynamic domains* in [Haynes87]) and utilize the symmetric shallow binding mechanism introduced in [Baker78c]. Their *states* are used to carry the *preludes* and *postludes* associated with *dynamic-wind*, a generalization of Common Lisp's `unwind-protect` primitive. While their preludes and postludes can be used to simulate our *actions* and *antidotes*, navigating between domains is not guaranteed to be atomic, and hence they cannot utilize domains for concurrency control or crash recovery.

The notions of multiple worlds originated with Kripke's models of modal logics, and come by way of Hewitt's Planner [Hewitt72], Rulifson's QA4 [Rulifson72] and Sacerdoti's QLISP. In fact, our worlds are concrete models of Pratt's "dynamic logic" of assignment [Pratt79]. McCarthy's *situations* [McCarthy69] are similar to our worlds, except that McCarthy makes situations explicit rather than implicit as in modal logics. The use of multiple worlds for synchronization in a design context is hinted at in Goldstein's *layers* [Goldstein81]. Curiously, global state worlds directly contradict our doctoral dissertation [Baker78a]; this contradiction is forced by the possibility of subverting a distributed system using fast communication. Our unificational mergings of multiple worlds are similar to the AND-parallelism of parallel Prologs, except that our processes work optimistically instead of waiting on unbound logical variables.

Our worlds are "mostly consistent", in the sense that any inconsistencies can be quickly listed. Alternative views of distributed systems [Tuttle90] assume that processes are "mostly inconsistent", and no attempt is made to reach unanimity during communication. These models are clearly more powerful, but are correspondingly more difficult to understand.

Multiple *versions* of objects apparently started as a file system concept, and were elevated to a model of concurrency control by Reed [Reed78]. *Virtual time* [Jefferson85] is an elegant use of multiple versions for optimistic concurrency control; Jefferson invents the notion of *antimessage*, which we have generalized into *antidote*. However, it doesn't seem to have occurred to anyone outside of the AI community to index versions by world instead of by object; indexing in this way finesses the garbage collection problems of versions.

Many implementation issues of multiple worlds have been studied in the context of variable-binding environments. Various schemes of *cacheing* [Teitelman74] and *shallow binding* [Greenblatt74] [Baker78c] have been studied. The multiple versions method for variable-binding environments was discussed in [Henhapl71], but was apparently not known to Reed and others studying concurrency control.

[Eswaran76] is the canonical reference for *transactions*, which terminate by either *committing* or *aborting*. [Kohler81] is an excellent reference on synchronization and recovery. *Logging* of transactions is the computer science generalization of journaling in double-entry bookkeeping systems. *Shadowing* is equivalent to our multiversion *update* operation implemented using either deep or shallow binding, depending upon whether the new page contains the new or old value.

Kung [Kung81] devised a optimistic protocol for database access which operates in three phases—read, validate and write. Our model easily simulates this three-phase protocol by building for each transaction a private cache for all accesses,[11] both reads and writes, and this cache becomes part of the transaction's current world. These caches allow for conflict resolution which is more intelligent than the simplistic "blind" commit-first protocol we have used. Since the "read lists" and "write lists" are encoded in the contents of the caches, the validate phase of the three-phase protocol can be emulated.

Our deep binding implementation of transactions is equivalent to *intention lists* [Lampson76], while our shallow binding implementation of transactions is equivalent to transaction logging for the purposes of UNDO. Our re-rooting operation is a generalization of DO/UNDO/REDO logs of System R [Gray81].

Knight [Knight86] invented the notion of "mostly functional programming", which is also advocated in [Baker90c].

[11]There is no possibility of cache inconsistency, because the objects cached can only be modified by the process which owns the cache.

REFERENCES

Ada83: *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Camb., MA, 1986.

Baker, Henry, and Hewitt, Carl. "The Incremental Garbage Collection of Processes". *Proc. ACM Symp. on AI and Progr. Langs., Sigplan Not. 12*,8 (Aug. 1977),55-59.

Baker, Henry. *Actor Systems for Real-Time Computation*. Ph.D. Thesis, MIT/LCS/TR197, March, 1978,145p.

Baker, Henry. "List processing in real time on a serial computer". *CACM 21*,4 (April 1978),280-294.

Baker, Henry. "Shallow Binding in Lisp 1.5". *CACM 21*,7 (July 1978),565-569.

Baker, Henry. "Unify and Conquer (Garbage, Updating, Aliasing ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, Nice, France, June, 1990,218-226.

Baker, Henry. "Shallow Binding Makes Functional Arrays Fast". ACM *Sigplan Not. 26*,8 (Aug. 1991), 145-147.

Baker, Henry. "Equal Rights for Functional Objects". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Bernstein, Philip A., and Goodman, Nathan. "Concurrency Control in Distributed Database Systems". *ACM Computing Surveys 13*,2 (June 1981),185-221.

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. "The notions of consistency and predicate locks in a database system". *CACM 19*,11 (Nov. 1976),624-633.

Goldstein, I.P., and Bobrow, D.G. "Layered Networks as a Tool For Software Development". *Proc. 7'th IJCAI*, (Aug. 1981),913-919.

Gray, Jim, McJones, Paul, *et al*. "The Recovery Manager of the System R Database Manager". *ACM Computing Surveys 13*,2 (June 1981),223-242.

Greenblatt, R. "The Lisp Machine". AI Working Paper 79, MIT AI Lab., Camb., MA, Nov. 1974.

Gruber, R.E. "Optimistic Concurrency Control for Nested Distributed Transactions". MIT/LCS/TR-453, June 1989,106p.

Habermann, A.N., and Nassi, I.R. "Efficient Implementation of Ada Tasks". TR CMU-CS-80-103, Carnegie-Mellon Univ., Jan. 1980.

Halpern, J.Y., and Moses, Y. "Knowledge and Common Knowledge in a Distributed Environment". *JACM 37*,3 (July 1990),549-587.

Hanson, Christopher and Lamping, John. "Dynamic Binding in Scheme". Unpublished manuscript, 1984.

Haynes, Christopher T., and Friedman, Daniel P. "Embedding Continuations in Procedural Objects". *ACM TOPLAS 9*,4 (Oct. 1987),582-598.

Henhapl, W., and Jones, C.B. "A run-time mechanism for referencing variables". *Inform. Processing Letters 1* (1971),14-16.

Hewitt, Carl. *Description and Theoretical Anaysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot*. Ph.D. Thesis, MIT AI TR-258, Camb., MA, April 1972.

Hewitt, Carl, and Atkinson, Russell. "Synchronization in Actor Systems". *Proc. ACM POPL 4* (Jan. 1977),267-280.

Hewitt, Carl, and Baker, Henry. "Actors and Continuous Functionals". In Neuhold, E.J. (ed.), *Formal Descriptions of Programming Concepts*. North-Holland, 1978, 367-390.

Hilfinger, Paul N. *Abstraction Mechanisms and Language Design*. MIT Press, 1983.

Hughes, G.E. and Cresswell, M.J. *An Introduction to Modal Logic*. Methuen and Co., London, 1968.

Jefferson, David R. "Virtual Time". *ACM TOPLAS 7*,3 (July 1985),404-425.

Katz, Morry. "Paratran: A Transparent, Transaction-based Runtime Mechanism for Parallel Execution of Scheme". MIT/LCS/TR-454, July 1989,84p.

Keller, R.M., and Lindstrom, G. "Toward Function-Based Distributed Database Systems". Tech. Rep. 82-100, Dept. of Computer Sci., U. Utah, Jan. 1982, 37p.

Knight, Tom. "An Architecture for Mostly Functional Languages". *Proc. 1986 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1986),105-112.

Kohler, Walter H. "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems". *ACM Computing Surveys 13*,2 (June 1981),149-183.

Kung, H.T., and Robinson, J.T. "On Optimistic Methods for Concurrency Control". *ACM Trans. Database Sys. 6*,2 (June 1981),213-226.

Lamport, L. "Time, clocks and the ordering of events in a distributed system". *CACM 21*,7 (July 1978),558-565.

Lamport, L., and Lynch, N. "Chapter on Distributed Computing". MIT/LCS/TM-384, Feb. 1989,60p.

Lampson, B.W., and Sturgis, H.E. "Crash recovery in a distributed storage system". Unpublished paper, Xerox PARC, Palo Alto, 1976.

McCarthy, J. and Hayes, P. "Some Philosophical Problems from the Standpoint of Artificial Intelligence". In Michie, D., and Meltzer, B., *eds.*, *Machine Intelligence 4*, Edingurgh Univ. Press, Edinburgh, Scotland, 1969.

Miller, James S. *MultiScheme: A Parallel Processing System based on MIT Scheme*. Ph.D. Thesis, MIT-LCS-TR-402, Sept. 1987,243p.

Osborne, Randy B. *Speculative Computation in MultiLisp*. Ph.D. Thesis, MIT-LCS-TR-464, Dec. 1989,263p.

Pratt, V.R. "Process Logic". *ACM POPL 6*, (1979),93-100.

Reed, David P. *Naming and Synchronization in a Decentralized Computer System*. Ph.D. Thesis, MIT EECS Dept., 1978.

Rich, Charles. "A Formal Representation for Plans in the Programmer's Apprentice". *Proc. 7'th IJCAI*, (Aug. 1981),1044-1052.

Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. "QA4: A Procedural Calculus for Intuitive Reasoning". SRI AI Ctr. Tech. Note 73, Nov. 1972,363p.

Sussman, G., and McDermott, D. "From PLANNER to CONNIVER — A Genetic Approach". *AFIPS FJCC* (1972).

Tuttle, Mark. *Knowledge and Distributed Computation*. Ph.D. Thesis, MIT/LCS/TR-477, May 1990,237p.

Ullman, Jeffrey D. *Principles of Database Systems*. Computer Science Press, Potomac, MD, 1980.

Weihl, W.E. "The Impact of Recovery on Concurrency Control". MIT/LCS/TM-382.b, Aug. 1989,22p.

Wiebe, Douglas. "A Distributed Repository for Immutable Persistent Objects". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),453-465.

© 1990, 1993 Nimble Computer Corporation