# System Support for "Multiple Worlds"

*Jonathan M. Smith*

*Gerald Q. Maguire, Jr.*

Technical Report CUCS-437-89†

Computer Science Department
Columbia University
New York, NY 10027

*ABSTRACT*

Concurrently computing alternative solutions to a problem can be used as a method to improve response time. In order to maintain internal consistency despite differences caused by alternative solution methods, "Multiple Worlds" are created. This paper examines the operating system requirements posed by such a method.

Problems include (1) side-effects and (2) combinatorial explosion in the amount of state which must be preserved. We solve these by process management and an application of "copy-on-write" virtual memory management. Side effects resulting from interprocess communication are handled by a specialized message layer which interacts with process management.

## 1. Introduction

A question which has intrigued many researchers is how an increasing supply of computational resources, in the form of multiple computers, can be utilized to solve bigger problems, to solve problems faster, and to solve problems more reliably. In [12] we discuss the approach of pursuing alternatives *concurrently*. The motivation for concurrent execution is an improvement in *response time*, which is the amount of wall clock time necessary to carry out a computation.

The typical paradigm for exploiting parallelism in a domain has been to decompose a problem into a number of cooperating processes. These processes are executed concurrently where hardware is available, e.g., multiprocessors.

In contrast, our "Multiple Worlds" scheme produces faster results using *competing* processes. The processes pursue alternative methods to a result, using a common input. When there are differences in the execution times, "Multiple Worlds" exploits this difference by picking the first process to complete and eliminating slower processes. The theoretical basis for the technique comes from order statistics. Consider independent identically-distributed random variables $\{X_1, \ldots, X_n\}$ which measure execution time and whose distribution function is $F(t)=Prob(X_i \leq t)$. We can compute $X^* = \min_{i=1}^{n}\{X_i\}$ which in practice is the random variable defined as the execution time of the fastest execution. A straightforward analysis shows that $F^*(t) \leq 1-(1-F(t))^n$. For an exponential distribution, $F(t)=1-e^{-\lambda t}$, $\bar{X}=\frac{1}{\lambda}$, and the variance $V=\frac{1}{\lambda^2}$. Thus, $F^*(t)=1-e^{-\lambda n t}$, so that $\overline{X^*}=\frac{\bar{X}}{n}$ and $V^*=\frac{V}{n^2}$. For large $n$, these results are independent of $F()$. We have implemented and measured several algorithms for which this statistical description correctly predicts the performance. The implications of the analysis are threefold: (1) the possibility for a linear speedup exists; (2) with faults represented by $X_i=+\infty$, the analysis holds, implying fault-tolerance; and (3) the reduction in variance has important implications for real-time systems.

Statistically independent alternatives can be the

† This paper has been submitted for publication. Please contact the authors for citation information.

result of randomized parameters or choices, multiprocessing workloads, interprocess communication, or input/output. The alternative solution methods can be either explicitly specified, or derived from nondeterminism inherent in a solution method.

Several interesting problems arise in support of this parallelism. The alternative solution methods may share state. To allow unrestricted access to this shared state, copies of the state are made, one for each alternative. So that each alternative can execute as if it was the sole solution method, "Multiple Worlds" are created to preserve this illusion. Much like the illusions of timesharing or virtual machines, the illusion requires significant system support. We discuss the nature of the system support in this paper.

## 2. Model

When several methods of computing a result are available, the existence can be stated as a set of *alternatives*, e.g.,

```
SELECT
   a
 OR
   b
 OR
   c
TCELES
```

a, b, and c comprise a block. The block's semantics are simple: one of its components is executed. In practice, we can optimize the block's response time with parallel hardware. This is done by executing the components concurrently. a, b, and c *compete*. The winner is selected by virtue of the fastest execution time. The other computations are aborted, in order to reduce the effect on throughput. So that the parallel transformation is transparent, the competitors are not allowed to interact. Maintaining this transparency at a reasonable cost in performance dictates many of the strategies employed in our design.

Use of other language features allows control of the execution, if necessary. For example, if one wanted to execute the alternatives in order a, b, c, loops and guards could be applied, e.g.,

```
FOR I=1 TO 3
  SELECT
    WHEN I=1: a
   OR
    WHEN I=2: b
   OR
    WHEN I=3: c
  TCELES
ROF
```

Only alternatives with open guards can complete. An error condition is raised when no alternative is successful.

### 2.1. Transparency

To preserve the sequential semantics, the strategy used is *copying*. Copying ensures that changes to shared state, e.g., state external to the block, will not conflict with changes made by another alternative. Copying also implies that special measures must be taken when copying changes the behavior, e.g., with semaphores. To simplify our discussion, we assume that all state can be categorized as either *source* or *sink* state. The division is made on the basis of idempotence; operations on *sink* devices can be retried without observable effects, while operations on *sources* cannot be retried. For concreteness, consider a page of backing store and a teletype device, respectively. Side effects which affect *sink* state can be hidden; this is a common technique in the implementation of such abstract operations as *transactions;* the idea is that the transaction has the property of *atomicity*, meaning that either none or all of the transactions component actions occur, and that intermediate states are not observable outside the transaction. Complex transactions may involve reads, which can occur unhindered, or writes, which must be done to a temporary copy until the transaction *commits*, or in other words, makes its changes permanent. Reads intended for the recently written copy are satisfied by that copy so that the transaction is internally consistent, i.e., it can read what it has written.

*Sink* state is manipulated as fixed-size *pages*. All sink state can be represented in this fashion; this is clear from implementations of a single-level store, as in MULTICS [9]. Thus we bury the entire memory hierarchy under the page abstraction; files are named sets of pages, and thus mechanisms which are used to transparently access files over networks (e.g., "Network File Systems") can be utilized to hide the network through the page management abstraction; an example is the Apollo DOMAIN Architecture [8]. When a computation is successful, its state changes are committed. The slower alternatives are easily discarded by halting their execution and returning their pages to the free pool. Thus, *undo* is extremely cheap and can be done asynchronously. Operations on *source* state must be blocked until successful completion of the alternative. These

operations could, in theory, be used to spur commitment, but that is not our strategy. An illustration of the problem is given by the following code fragment; `stdin` and `stdout` are mapped to a *source*.

```
SELECT
  {
    scanf( "%d", &num ); printf( "got %d\n", num );
  }
 OR
  {
    scanf( "%s", name ); printf( "name was: %s\n", name );
  }
TCELES
```

It should be clear that buffering strategies as suggested in Cooper's thesis [2] are ineffective here. Since the computations are not replicas, we cannot assume that they will perform the same operations in the same order. Thus, even the clever buffering scheme employed in Jefferson's "Time Warp Operating System" to make rollback transparent is inadequate.

We have chosen to block operations on source state until commitment. Choosing the first computation to perform operations on source state reduces the burden on the process scheduling algorithm, but may reduce our discrimination in choosing the fastest process.

### 2.2. System Structure

A *process* is an independently schedulable stream of instructions. Each alternative executes as a process. Therefore, as in UNIX, a close relation between a thread of control and an address space is enforced.

Processes are provided with a virtual machine, whose architecture is composed of machine instructions and a set of kernel primitives. The primitives may use instructions not included in the virtual machine architecture.

Interprocess communication (IPC) is accomplished through message-passing. Thus messages serve as the only means of propagating state changes. We assume that lower layers of the IPC protocol hierarchy enforce reliability (e.g., no lost or duplicated messages) and message ordering.

## 3. Implementation of "Multiple Worlds"

Two primitives, `alt_spawn()` and `alt_wait()` are used to implement the multiple worlds implied by the system model. The caller of `alt_spawn()` is a process, hereafter referred to as the *parent*, which contains a block of alternatives. The primitive is passed a positive integer, n, and if successful, it creates n copies of the caller. `Alt_spawn( n )` returns 0 to the parent and numbers from 1 to n in the copies. Thus, `alt_spawn( 1 )` can be used to implement the familiar *fork()* primitive. `Alt_spawn()` also allocates a semaphore, whose use will be seen later. Code generated by the appearance of alternatives in a program would be (in pseudo-C syntax):

```
switch( alt_spawn( n ) )
    {
    case 0: /* parent */
        alt_wait( TIMEOUT );
        fail(); /* if returned */

    case 1: /* First alternative *
                .
                .

                .
                .
    case n: /* n-th alternative */
                .
                .
        alt_wait( 0 );
    }
```

`Alt_wait()`, as illustrated in the example, is called upon completion of the alternative's code. The alternative, hereafter referred to as a *child* process, calls `alt_wait()` with an argument of 0, indicating that the process wishes to complete immediately, if possible. The parent process passes `alt_wait()` a TIMEOUT value, which is used to determine failure of the block. Pseudo-code for `alt_wait()` follows:

```
if( Parent ) {
  wait( TIMEOUT time units );
  deallocate semaphore;
  terminate children;
  return error indication;
}
else {
  wait( TIMEOUT time units );
  if no semaphore exit;
  deallocate semaphore;
  terminate Parent;
```

```
    inherit Parent's process id;
    terminate siblings;
    return;
}
```

TIMEOUT is typically non-zero in the parent, as TIMEOUT represents the time the parent is willing to wait for a successful child call to `alt_wait()`. TIMEOUT's value should be chosen so that after TIMEOUT time units have elapsed, it is unlikely that any of the alternatives have succeeded. While choosing such a value is very hard, most computations have an execution time which is clearly unacceptable to the application; this value can then be used.

Thus, the flow of control through the child appears to have been seamless, up to and including maintenance of the process id.

Alternatives which are guarded can evaluate the guard condition and exit upon failure; such alternatives never call `alt_wait()`. Such guards could also be evaluated before `alt_spawn()` is called, improving throughput at the expense of response time. The guard can also be reevaluated, for redundancy.

When the first child invokes `alt_wait()`, commitment of its results occurs. Its parent and sibling are terminated; they have speculated that they will complete first, and are now known to be wrong. The primitive returns for only one process, and thus serves to choose 1 process from as many as n+1 processes.

The use of the semaphore is "barrier" synchronization, so that sibling elimination is robust. In a distributed setting, this semaphore can be implemented as suggested by Schneider [10]. It provides additional safety when process termination is unreliable, and can be made fault-tolerant through use of a consensus strategy.

### 3.1. Interprocess Communication

Our scheme must maintain transparency, and alternatives may communicate with other processes. State changes which an alternative causes via a message cannot be propagated to source state, or we might create inconsistency, as was illustrated with the I/O interaction in section 2. However, interactions which could be carried on entirely with sink state should not cause blocking.

A *message* from process P to process S has the following three part structure:

1) A sending *predicate*, encapsulating the assumptions under which the *sender*, P, sends the message.
2) The data comprising the message.
3) Some control information, e.g., sender id, destination id, etc.

Each process created by an `alt_spawn()` call has attached to it a set of predicates. This set is composed of whatever predicates were attached to the parent (to allow nesting), plus a new completion predicate. The completion predicate encapsulates the process's assumption that it will complete successfully. These predicates are stored as lists of process identifiers. Where `a` and `b` are siblings from the same `alt_spawn()`, `complete(a)` implies ¬`complete(b)`. Such predicates share simplicity of management with schemes such as Eswaran, et al.'s [4]. Process predication versus data predication is a simple choice, as processes change state much less frequently than they access data. Updated pages are predicated by virtue of their ownership by a predicated process. When a receiving process accepts a message, its predicates ( **R** ) are checked against those attached to the message ( **S** ). If the assumptions that the receiver makes about the "state of the world", as encapsulated in the predicates, agree with those of the sender (e.g., $\mathbf{S} \subseteq \mathbf{R}$ ), the message is immediately accepted.[1] If the receiver's predicates conflict ($p \in \mathbf{S}$ and $\neg p \in \mathbf{R}$), the message is ignored, and if the receiver must make further assumptions to accept the message ( $p \in \mathbf{S}$ and $p \notin \mathbf{R}$), two copies of the receiver are created. Define `complete(P)` to be `TRUE` when process `P` successfully synchronizes with its parent process, `FALSE` when `P` has assumed ¬`complete(Q)` for some process `Q` for which `complete()` has become `TRUE`, and otherwise indeterminate. One of the two copies is created with the predicates set to the previous values in conjunction with `complete(S)`[1]; the other is set up with its predicates as before, except that `complete(S)` is negated.[2] This is easy given the representation as two lists (i.e., "must complete" and "can't complete") of process identifiers. When the sending process succeeds or fails, one of the two receivers must be eliminated in order to maintain a consistent "state of the world;" at this point the additional assumptions which receipt of the message caused will become `TRUE`, and they can be eliminated from the lists.

[1] Thus implying all the sender's predicates.

[2] Thus implying rejection of the sender's predicates without creating a logical impossibility. Assuming the negation of *all* of S's predicates might imply that two mutually exclusive processes must complete.

To illustrate the idea, consider a group of communicating processes composed of P, Q, and S. P has children a, b, and c; Q has children d, e. f. This is illustrated by figure 1.

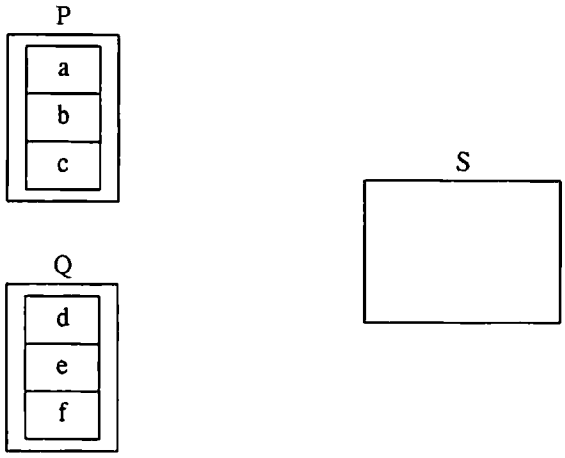


**Figure 1:** Communicating processes

Suppose both b and f wish to communicate with S. We use the predicates to create multiple copies of S, with which the alternatives communicate. This is illustrated in figure 2.

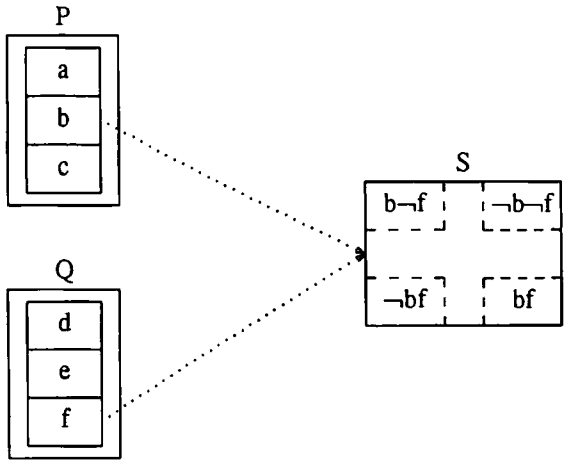


**Figure 2:** Communicating processes after message receipt

There are several implications to this scheme:

1. There is clearly a potential for a combinatorial explosion in the number of processes that exist, and amount of state which must be copied. Since the process creation is based on message receipt, it is "lazy".
2. Commit is *very* fast, as it can accomplished using only pointer (page descriptor) manipulation. For example, ¬bf can be committed by simply updating the page descriptor table associated with S.
3. Messages must be duplicated, as all valid receivers must get the message.
4. The method is *optimistic*. Rather than locking a resource based on a predicate, the assumption embodied in the predicate is sent with messages and used to create new "worlds."

While a process has predicates which are unsatisfied, it is restricted from causing observable side-effects, and thus cannot interface with *sources*.

The net effect is similar to the *atomicity* required of *transactions*, in that intermediate portions of the computation are not observable. A predicated process can however continue to receive messages, thus a process can be used to buffer data destined for source state.

## 4. Performance Analysis

The introduction's argument promises linear speedup. There are two facts which frustrate this promise. First, the dispersion of execution times must be significant, and the probability distribution function, $F()$, must have a long "tail." Second, we ignored the overhead involved in concurrent execution such as copying, sibling elimination, and processor contention.

To estimate the performance of the scheme, we can calculate

$$\frac{\textit{Expected time for sequential execution}}{\textit{Expected time for parallel execution}}$$

which for random input distributions can be estimated by

$$\frac{\textit{Mean execution time of alternatives}}{\textit{Smallest execution time + Overhead}}$$

The estimate is affected by both the dispersion (ratio of mean to best) and the relative overheads (ratio of mean to overhead). Since the dispersion is determined for the most part by the application, we have chosen to focus on reducing the overhead.

Extensive measurements have shown that sibling elimination is remarkably inexpensive. For example, elimination of 16 processes requires about 4 milliseconds per process on a Hewlett-Packard HP9000/350. This time is quite insensitive to variables such as process size and run-time behavior (e.g., CPU-bound). Broadcast capabilities and asynchronous elimination reduce the per-process time to about 2 milliseconds. Since termination is a naturally parallel task, execution time will be further reduced on parallel architectures.

Processor contention can be a serious problem, but there is a simple approach: we limit the number of

alternatives to the number of processors. In practice, the overhead implied by copying bounds the number of processors which can be applied.

Experiments with process migration software [13] we wrote for extension of "Multiple Worlds" to a distributed setting indicated that the dominant overhead is *copying*. The benefit of lazy copying [1] is enormous. In Smith and Maguire [11] we quantified the advantage of "copy-on-write" *fork* operations under UNIX. Our measurements were made on two workstations, the AT&T 3B2/310 and the Hewlett-Packard HP9000/350. For the 3B2, a *fork()* (with no memory updates to a 320K address space) takes about 31 milliseconds; under the same conditions the HP requires about 12 milliseconds. The measured service rate of page copying was 326 2K pages/second for the 3B2, and 1034 4K pages/second for the HP. The fraction of the pages in the address space which are written is the important independent variable for a program with a known address space size, using "copy-on-write." These costs should be representative of a shared memory configuration of equivalent processor technology.

Sophisticated migration schemes using "on-demand" state management techniques have been constructed [15, 17]. The crucial parameter is *locality of reference*, which most programs exhibit.

Using lazy copying techniques reduces the overhead of copying dramatically, and hence reduces the penalty of this overhead on execution time.

## 5. Applications

In related work [12] we have discussed OR-Parallelism in Prolog and distributed execution of recovery blocks as applications for "Multiple Worlds". Here we study an application from numerical analysis.

Finding the zeros of a polynomial is a classic problem in numerical analysis. An algorithm is passed a list of coefficients, and returns a list of values at which the polynomial evaluates to zero, if successful. One widely-used algorithm for finding complex zeros of polynomials with complex coefficients is the Jenkins-Traub algorithm [6], ACM algorithm #419. When the algorithm finds a root, it *deflates* the polynomial and attempts to solve the deflated polynomial. This process continues until all the roots are found. The roots must be found in order (approximately) from smallest to largest for the deflation process to be numerically stable. Thus, other than potential for vectorization, the algorithm is inherently sequential. An additional constraint in numerical applications is that re-ordering operations can perturb results. Algebraically sound reorderings are dangerous due to the non-commutativity of floating point arithmetic on finite-precision machines.

The algorithm converges extremely rapidly in the neighborhood of a zero. Like many numerical iterations, an appropriate starting value is necessary for convergence. The starting value in this case is a point in the two-dimensional complex plane. For polynomials, a lower bound on the complex modulus of the smallest root can be calculated. This bound defines a circle of potential starting values. A point on this circle, determined by an angle, is chosen at random to start the search. If the rootfinder fails on this starting value, a new angle is chosen; the algorithm fails if a fixed number of rotations do not result in a successful starting value. In practice, empirically determined values for the starting angle and rotation are used. We exploited this non-deterministic choice of starting values for an impressive speedup using "Multiple Worlds."

For our experiments, we transliterated the published algorithm from FORTRAN to C and regression tested to ensure correctness of the recoding. We added an additional parameter to the zerofinder to pass it an angle for starting and rotation. Our first set of experiments found that the execution time of the algorithm was sensitive to the choice of angle on many polynomials, particularly those which are of large degree or ill-conditioned. This encouraged parallel execution. For $N$ processors, we divided the circle into $N$ arcs of length $\frac{2\pi}{N}$. A point on each arc is chosen as a starting value.

To emulate `alt_spawn()`, one UNIX process per angle is spawned using *fork()*. Each process attempts to solve the polynomial, while the spawning process waits for completion of an alternative. When a process completes the zerofinding successfully, it attempts to acquire an exclusive semaphore. If the attempt fails, the process exits, as another process has preceded it to this barrier. If the lock is acquired, the process terminates its siblings and parent with a broadcast signal issued with the UNIX *kill()* primitive. This sequence of actions serves to emulate `alt_wait()`; the UNIX implementation of "copy-on-write" forks ensures that the complete state of the parent process is available to the successful alternative.

"Multiple Worlds" wins when the execution time is less than the average of the times for the $N$ angles. While we've tested a variety of polynomials to remove spurious results, we present only a representative example here. The degree 16 polynomial defined by the coefficients

$$z^{16}+$$

$$(19.8-3.2i)\cdot z^{15}+$$

$$(176.93-58.94i)\cdot z^{14}+$$

$$(940.246-494.724i)\cdot z^{13}+$$

$$(3284.8747-2495.183i)\cdot z^{12}+$$

$$(7821.1515-8375.77856i)\cdot z^{11}+$$

$$(12521.112583-19491.045162i)\cdot z^{10}+$$

$$(12045.424807-31405.685381i)\cdot z^{9}+$$

$$(2785.185034-32740.983162i)\cdot z^{8}+$$

$$(-10378.708048-15172.462694i)\cdot z^{7}+$$

$$(-16870.986162+13554.317038i)\cdot z^{6}+$$

$$(-12430.788642+33571.308326i)\cdot z^{5}+$$

$$(-3673.995257+33389.404113i)\cdot z^{4}+$$

$$(1360.752190+20180.264386i)\cdot z^{3}+$$

$$(1689.471245+7674.191752i)\cdot z^{2}+$$

$$(622.122192+1700.277923i)\cdot z^{1}+$$

$$(86.407858+168.238421i)$$

is sufficiently troublesome that several choices of angle cause failure of the algorithm; the potential for fault-tolerance of the algorithm is clear from this example. The processes were executed concurrently on an experimental multiprocessor workstation with a "closely-coupled" architecture; the memory access is non-uniform [3]. One portion of each processor's address space is mapped to a shared global memory, while another portion is mapped to a private "local" memory. The idea is that the global memory will be used for fast communication and the local memory serves as a cache to reduce bus contention. The operating system is CMU's MACH [16], which emulates UNIX. The results are shown in table I, all times in seconds:

| procs | max | min | avg | fails | par |
|---|---|---|---|---|---|
| 1 | 12.1 | 12.1 | 12.1 | 0 | 12.8 |
| 2 | 13.1 | 6.2 | 9.6 | 0 | 6.3 |
| 3 | 22.9 | 12.1 | 17.1 | 1 | 12.3 |
| 4 | 13.4 | 6.2 | 10.7 | 0 | 6.4 |
| 5 | 19.9 | 5.8 | 12.2 | 0 | 6.3 |
| 6 | 23.0 | 6.2 | 13.8 | 1 | 11.4 |
| 7 | 16.9 | 5.9 | 10.1 | 0 | 7.4 |

Table I: Jenkins-Traub performance with "Multiple World

The first column, labeled **procs**, indicates the number of processors applied to the problem. Sequential execution on a single processor was used to determine the worst, best, and average times used by the algorithm. These values are shown in columns **max**, **min**, and **avg**, respectively. These times are CPU times, and do not show any delays or system overhead; the accuracy is limited by the clock granularity. The **fails** column indicates the number of angle choices for which the algorithm failed to find all of the roots. The **par** column shows the time for parallel execution measured using wall-clock execution time. Thus, any overhead incurred by the execution scheme will be reflected in this difference. The differences between **min** and **par** can be used to estimate the overhead; in the 2 processor case it's about 0.1 second, a small fraction of the total time. Significantly, the parallel execution always did better.

It's clear from the tabulated results that the speedup is not proportional to the number of processors, as it's data-dependent. However, the speedup was achieved on a problem with which conventional parallel execution techniques have difficulty. In addition, the zerofinding process is now somewhat fault-tolerant; failures were tolerated in the 3 and 6 processor cases. The portion of the executable's address space which is read-only is greater than 75 percent.

Thus, we have used the execution time variance caused by random choice of a starting value to achieve a speedup. The overhead in this example was low, and we used a simplistic scheme to exploit the random choice available. How aggressively available parallelism is exploited in general is a function of the overhead associated with maintaining a process. However, once this is known, the proper granularity can be used as a factor in the decomposition process.

## 6. Related Work

Kung [7] examined the advantages of concurrently executing processes which were very loosely synchronized. Kung's processes cooperated, and hence occasionally communicated results using shared

memory. As side effects do not present the major problems they do in our setting, their management was not addressed.

Jefferson, et al., have proposed and implemented a "Time Warp Operating System," [5] (TWOS) which employs the notion of Virtual Time. The basic idea of the system is that all the state changes take place because of *messages*, and with deterministic processes, a set of *relative* time constraints imposed by message order determine the behavior of the system. Processes consume messages in their queues eagerly, implying optimistic computation. When messages arrive out of "Virtual Time" order, the process has violated a synchronization constraint and is "rolled-back" using a stored checkpoint. Message "undo"s are accomplished with timestamped "antimessages."

While the processes typically cooperate, the system takes advantage of asynchronous execution and scheduling opportunities for impressive speedups on problems such as large simulations. Since there is rollback, TWOS has to cope with side-effects to source state in much the same way as "Multiple Worlds". In practice, this is accomplished through use of a special-purpose buffering process called *stdout*. An important observation is that TWOS corrects mistaken assumptions by rolling back, which may delay progress. "Multiple Worlds" corrects mistaken assumptions by *elimination;* another alternative is presumed successful.

Exploring alternatives in parallel is far from a new idea; hardware engineers looked to it as a way of maintaining pipeline utilization in early high-speed computers, such as the IBM 360 Model 91. Their approach was to prefetch components of both possible branch paths until either the results of the conditional execution are available (in which case the correct stream can be chosen and the other discarded) or an irreversible side effect (such as instruction execution) would occur. Our management of side effects lets us proceed further, as a great deal of computation can occur before the side effects become observable at commitment.

Our method uses simple predicates to detect conflicts, but delays their resolution as long as is possible. Thus, it is *optimistic* in the sense that each timeline assumes that it will succeed. At each point where this success may come into question, it generates a predicate. Thus, there is as little *waiting* as possible in the system, e.g., for *locks*. In other settings, such methods are called *optimistic* [14] because they assume that delay-causing or failure-causing conditions happen infrequently. Thus, normal operation is made cheap, at the expense of somewhat more expensive handling when the assumption is wrong. In our setting, the operant *optimistic* assumption is that the executing alternative is the one which will complete successfully. Thus, the predicates indicate that a process assumes that it will complete successfully; rather than *waiting*, it *continues under that assumption.* This works in our case because some alternative is already pursuing the recovery strategy; thus, there is no execution time penalty paid for recovery.

The notion of multiple alternatives is orthogonal to the transaction concept; if we view an alternative "block" as effecting a transaction on the system state, the specification is a description of how to accomplish the transaction reliably. Alternately, "Multiple Worlds" could be viewed as a set of "competing" transactions, at most one of which will take effect.

Distribution of computation across several nodes offers attractive possibilities for both reliability and performance. Cooper [2] discusses the use of replicated distributed programs in order to take advantage of this potential. Cooper's CIRCUS system transparently replicates computations across several nodes in order to increase reliability. Replication is somewhat different than the problem we have examined, mainly because we cannot depend on all of the concurrent alternatives exhibiting the same behavior, e.g., reading and writing. For example, when managing I/O for replicated computations, only one read operation can be performed, and its results buffered for subsequent readers of the same data. Thus, idempotency of some *source* state can be forced through buffering. Transparent replication can easily be combined with the use of parallel execution of several alternatives for increases in performance, reliability, or both.

## 7. Conclusions

The `alt_spawn()` and `alt_wait()` primitives are used to support competing "Multiple Worlds" whose timelines proceed in parallel. They are similar to a multiway *fork()* and multiway *join()* respectively. `Alt_spawn()` is optimized by lazy copying with "copy-on-write." `Alt_wait()` reduces the effect on system throughput by eliminating all processes other than its first caller.

While we have not completed an implementation, the effectiveness of the scheme was tested with a prototype constructed using MACH. We used a polynomial zerofinding on a multiprocessor. "Multiple Worlds" showed speedup and fault tolerance. The

overhead added by the scheme amounted to a small fraction of the execution time. While copying costs are significantly higher in a distributed setting, "Multiple Worlds" will remain effective for a large class of computation-intensive applications. Since the concurrent constituents do not cooperate, synchronization costs are reduced. `Alt_wait()`'s barrier synchronization can be achieved with a distributed semaphore.

In summary, we've shown that the use of "Multiple Worlds" to execute computations competitively is a viable alternative to the traditional cooperative model for parallel processing. The "Multiple Worlds" scheme ensures that this performance improvement is achieved in a manner which is transparent to the application programmer. Since neither approach precludes use of the other, future work should explore combining the two strategies. Further experimental work will also provide insight on the performance implications of the interprocess communication scheme.

## 8. Notes and Acknowledgments

Rob Strom has inspired and helped refine many of the ideas presented here. Discussions with Calton Pu, Steve Feiner and Dave Farber have refined our ideas. Yechiam Yemini pointed out the statistical analysis of speedup potential.

This work was supported in part by equipment grants from the Hewlett-Packard Corporation and AT&T, and NSF grant CDR-84-21402.



## 9. References

[1] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communications of the ACM* 15(3), pp. 135-143 (March 1972).

[2] Eric Charles Cooper, "Replicated Distributed Programs," Ph.D. Thesis, University of California, Berkeley (1985).

[3] IBM Corporation, *A Brief Users Introduction to the ACE Multiprocessor*, 1988.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM* 19, pp. 624-633 (November 1976).

[5] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Time Warp Operating System," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 77-93, In *ACM Operating Systems Review 21*:5 (8-11 November 1987).

[6] M. A. Jenkins and J. F. Traub, "Algorithm 419: Zeros of a Complex Polynomial," *Communications of the ACM* 15, pp. 97-99 (February, 1972).

[7] H. T. Kung, "Synchronized and Asynchronous parallel Algorithms for Multiprocessors," in *New Directions and Recent Results in Algorithms and Complexity*, ed. Traub, J. F. (1976). Academic Press

[8] D.L. Nelson and P.J. Leach, "The Architecture and Applications of the Apollo Domain," *IEEE Computer Graphics*, pp. 58-66 (April 1984).

[9] Elliott I. Organick, *The Multics System*, Massachusetts Institute of Technology Press (1972).

[10] Fred B. Schneider, "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems* 4(2), pp. 179-195 (April 1982).

[11] Jonathan M. Smith and Gerald Q. Maguire,Jr., "Effects of copy-on-write memory management on the response time of UNIX *fork* operations," *Computing Systems* 1(3), pp. 255-278 (1988).

[12] Jonathan M. Smith and Gerald Q. Maguire,Jr., "Transparent Concurrent Execution of Mutually Exclusive Alternatives," in *Proceedings, Ninth International Conference on Distributed Computing Systems*, Newport Beach, CA (June, 1989).

[13] Jonathan M. Smith and John Ioannidis, "Implementing remote *fork()* with checkpoint/restart," *IEEE Technical Committee on Operating Systems Newsletter*, pp. 12-16 (February, 1989).

[14] R. E. Strom and S. Yemini, "Synthesizing Distributed and Parallel Programs through Optimistic Transformations," in *Current Advances in Distributed Computing and Communications* (1987). Computer Science Press

[15] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," in *Proceedings, 10th ACM Symposium on Operating Systems Principles* (1985), pp. 2-12.

[16] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*,

Austin, TX, pp. 63-76, In *ACM Operating Systems Review* 21:5 (8-11 November 1987).

[17] E. Zayas, "Attacking the Process Migration Bottleneck," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 13-24, In *ACM Operating Systems Review* 21:5 (8-11 November 1987).