Exploring "Multiple Worlds" in Parallel

Jonathan M. Smith

Gerald Q. Maguire, Jr.

Technical Report CUCS-436-89†
Computer Science Department, Columbia University, New York, NY 10027

ABSTRACT

We examine computing alternative solutions to a problem in parallel to improve response time.

Problems with exploring multiple alternatives in parallel include (1) side-effects and (2) combinatorial explosion in the amount of state which must be preserved. These are solved by process management and an application of ''copy-on-write'' virtual memory management. The side effects resulting from interprocess communication are handled by a specialized message layer which interacts with process management. The effect is to create ''Multiple Worlds'', which are different due to the alternate solution methods, but internally self-consistent.

We show how such a ''Multiple Worlds'' scheme can be used for several applications.

1. Introduction

> *"They could be different. For instance, the 1939 that exists 'now' back up the timeline might not contain a Hitler at all. When it arrives at its own 1945, World War II won't have happened, and it will have evolved a history that doesn't read like ours at all. From there it will go on into its own future, fully consistent with its own part but different than ours."* [9]

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. We examine a specific computational problem here, that of pursuing alternatives. Our designs show what can be done in order to execute instances of this problem type, speculatively, in parallel.

We are interested in what performance gains can be achieved. We measure *performance* using the metric of execution time, which is the amount of wall clock time necessary to carry out a computation. Thus, we may increase performance by this measure, while decreasing performance by measures such as *throughput.* Given this bias, we may risk wasted work in *speculative* computation, which throughput-oriented performance measures would discourage.

We begin by describing the computations to be analyzed. These are essentially a set of alternative methods for causing a state change to take place, with the additional constraint that at most one of the alternative state changes occurs.

In this paper, we review our problem setting; show a transformation which allows alternatives to execute concurrently while preserving sequential semantics; analyze the performance over complete problem domains as opposed to a point measure. Example application areas for our method are presented, and initial performance results on a numerical problem are provided.

1.1. Sequential Model

Our sequential model is as follows [20]. Several alternative methods of computing a result are available. Some of the alternatives may compute an acceptable result, while others may not. The essential problem is the choice between successful alternatives, or an indication of failure if there are no such alternatives.

What we want is for at most one of the alternative methods to be applied to our problem, or for whatever conditions constitute failure to be indicated. Each method, $1, \ldots, n$, has associated with it a *guard* condition, which it must satisfy in order to be considered successful. Each method is called an *alternative.* When the alternatives are composed into a block, the

† This report appears in the Proceedings of the 1989 International Conference on Parallel Processing. Please cite that version.

meaning is that one of the alternatives (including failure) are selected non-deterministically; this selection is the result of the block. The non-determinism in selection is necessary for higher-performance computing; see [21] for a more complete discussion on exploiting randomness with parallelism. The selection is non-deterministic and *unfair,* in that the selection of alternatives is not equiprobable, and should not be; it's clear that the alternative of failure should be given as low a probability of success as is possible, noting that when all the alternatives fail its conditional probability must be 1.

2. Parallel Execution

> *"This picture implied some parallel branching structure of universes in which every point along a timeline became a branch-point into a possibly infinite number of other timelines, with the branches forking unidirectionally like those of a tree."* [9]

2.1. System Model

A *process* (P) is an independently schedulable stream of instructions. In implementations, it is often associated with some unit of state, e.g., an address space, and a set of operations provided by a *kernel* to manage that *state.* Interprocess communication is accomplished solely through passing *messages.* Thus, a *message* is the only means by which:

- P_m can make P_j aware of a change in P_m's state.
- P_m can cause a change in P_j's state.

Interprocess communication (IPC) is assumed to behave reliably (no lost or duplicated messages) and FIFO (no out of order messages).

System *state* is divided into two types, *sink* and *source.* 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 was 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 [14]. 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 [13].

2.2. Process Management

Two primitives are used for process management. Process management creates, schedules, and terminates the mutually exclusive (and oblivious) alternatives. To spawn the alternatives, the parent uses `alt_spawn( n )`, which returns numbers from 1 to n in the alternatives and 0 to the parent. Thus a language preprocessor applied to a program with mutually exclusive alternatives would generate (in pseudo-C):

```
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 );
}
```

The functions of `alt_wait()` are manifold; the purpose is synchronizing in order to establish a single path through the tree of possible computations. The taken path is reflected in the execution history of the running process. `Alt_wait()` takes a TIMEOUT value as an argument. It 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.

When a spawned alternative calls

alt_wait() at the termination of its computation, a rendezvous between the alt_wait()ing parent and the child is effected. The parent is waiting, because if it was executing, it could cause state changes which would make its state inconsistent after the synchronization discussed in the next section. The behavior is much like that of the UNIX exec() system call, which overwrites the calling process and begins execution of the called process. In the case of the parent's call to alt_wait(), the parent process absorbs the state changes made by its child by atomically replacing its page pointer with that of the child (some copying might be needed for efficiency in the distributed case). Thus, the flow of control through the child appears to have been seamless, up to and including maintenance of the process id.

Use of these primitives is shown by concurrent execution of alternative methods shown in figure 1:

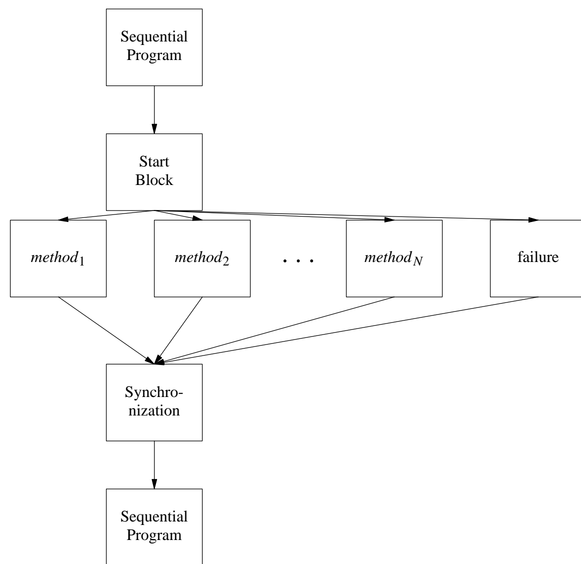


Figure 1: Concurrent Execution of Alternatives

Assuming that all the GUARD conditions have been satisfied, a process which completes its program segment attempts to synchronize. If any of the conditions required by the GUARD were not satisfied, the process aborts without synchronizing. Note that the GUARDs can be executed serially before spawning the alternatives (thus improving throughput at the expense of response time); in the child process; at the synchronization point; or at any combination of these places, for redundancy.

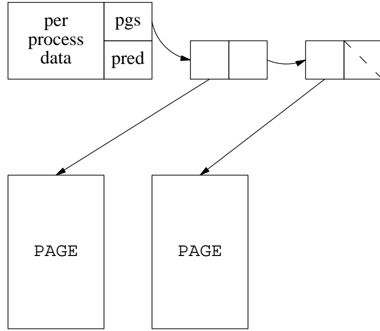
2.2.1. Synchronization

The synchronization point is where the first alternative child invokes alt_wait(). Alt_wait() is an "at most once" operation for any group of child processes created by the same alt_spawn(). In order to minimize the effect on throughput, when an alternative is selected, its "siblings" are eliminated. This is done by informing the scheduler that the processes are to be terminated. The deletion can be accomplished synchronously (where the other alternatives are deleted before execution resumes in the parent) or asynchronously (where the deletion occurs at some time after the alt_wait() resumes in the parent, but exactly when is not specified); experiments (mentioned at the end of section 3) indicate that asynchronous elimination gives better execution-time performance, once again at the expense of throughput.

2.3. Predicates

Ideally, we would like an alternative to carry on with its computation as much as it can before either blocking or synchronizing. In order to effect this, we add "predicates" to the messages. The predicates are lists of process identifiers, some of which the sending process depends on completing successfully and others on which the sending process depends on to *not* complete successfully. Thus, these are even simpler and easier to manage than the predicates described by Eswaran, *et al.* [7] The advantage of this representation over predication of data objects is that we can update the value of these elements as processes change status (e.g., running, blocked), with the idea that processes change status much less frequently than they make memory references to objects. These lists are constructed in two ways. First, the predicates of a "child" process consist of those of the "parent"; this allows for nesting and potentially complex dependencies. Second, when the "parent" spawns each of its alternative "children", each of the children additionally assumes that it will complete successfully, and that its siblings will not; thus "sibling rivalry" is taken to its extreme in this design! The failure alternative assumes that none of the siblings will complete. The state management strategy is "copy-on-write" [3] with page map inheritance from the parent, thus it is easily implemented within the context of a system which provides such features, e.g., CMU's MACH, and benefits from existing hardware support, e.g., for the WE® 32101 MMU [2]. The software-implemented predicates are used in the process control and message transmission activities to maximize sharing. Updated and newly-written pages are

predicated by virtue of their residence in a per-process descriptor table, as illustrated in Figure 2.



2.4. Interprocess Communication

2.4.1. Messages

A *message* from P_m to P_j has the following three part structure:

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

Each *process* in a *multiprocessing* (e.g., timesharing, multiprocessor, or distributed) system has a *unique identifier*, used to identify the process both within the system (e.g., for scheduling and resource allocation), and further, for interaction with other processes.

2.4.2. Multiple Worlds

An idea from science fiction, inspired by Dewitt's [6] multiple worlds notion, is appropriate here. The problem with interprocess communication stems from the fact that a given alternative may or may not be successful. In the case where it is successful, its execution results are available to the calling process. Where it is not successful, its results and any side-effects it may have generated must not be observable. These include side-effects due to interprocess communication.

The message system, the virtual addressing mechanism, and the process management mechanism are linked in the following way. When a receiving process accepts a message, its predicates ($\mathbf{R}$) are checked against those attached to the message ($\mathbf{S}$). 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. 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)` (thus implying all the sender's predicates) the other is set up with its predicates as before, except that `complete(S)` is negated, 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! This is shown in a revision of a previous figure:

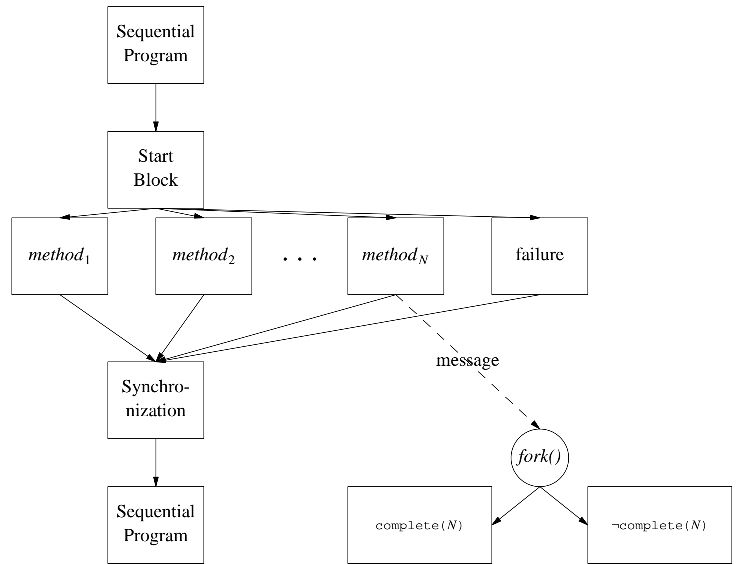


Figure 2: Use of predicates

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

3. Performance Analysis

The possibility of a performance increase stems from the fact that we can select the fastest alternative by means of the synchronization protocol. The cost we must pay for obtaining execution time proportional to the time for the fastest alternative is use of available hardware.

The effects of various overheads and system parameters are analyzed in the next section.

3.1. Overhead

In order to understand the overhead implied by the method, we can compare it to the best-case sequential execution of the fastest alternative. The penalties we are paying for parallel execution of all alternatives versus sequential execution of a selected alternative are

1. Memory Copying. In the distributed case we must actually copy state for a remote child so that the child can read or write locally. In the shared memory multiprocessor case, the copying overhead (in execution time) is reduced as the interprocessor bandwidth is much higher, and the latency is much lower. Even if the interprocessor bandwidth increases, latency will still restrain distributed performance. There is more copying to be performed during synchronization, as the changed state is updated in the parent's storage. The parent is constrained to remain blocked while the children are executing.
2. Sibling elimination. This is can be done asynchronously with respect to continuation of the selected alternative, and is naturally parallel, but the instructions to terminate the alternatives must still be issued, and they increase with the number of alternatives.

3.2. Analytic Description

We have provided a more detailed description of the setting for our performance analysis in Smith [20]; here we present an abbreviated version. Assume that we have N alternative methods of performing a *computation,* $C_1, \ldots, C_N$. We have also a measure of clock time, τ, for $C_i(\vec{x})$, $\tau(C_i, \vec{x})$ is the time required for C_i to compute a result given input $\vec{x}$. How can we use the availability of these alternatives to lessen our execution time? If we know, through analysis or empirically, that one of the methods is always faster than a second, we discard the second from consideration. If the relationship between the performances is less predictable, there are other possibilities:

A. Statistical data can be applied, e.g., *quicksort* is "almost always" $O(n \log n)$. Thus, we'll rarely go wrong to use it.

B. An algorithm can be selected at random from amongst the C_i when given $\vec{x}$.

C. The C_i can be applied to $\vec{x}$ concurrently; the first C_i which produces an acceptable output is selected. The other C_i are irrelevant and can be terminated.

Scheme **A** relies on information which may not be available. Scheme **B**, when run repeatedly on some input $\vec{x}$, will perform at the arithmetic means of the computations' performance, i.e., $\frac{\sum_{i=1}^{N} \tau(C_i, \vec{x})}{N}$. It is interesting to note, as well, that failures or infinite loops will frustrate Scheme **B**. For notational convenience, we define an artificial algorithm C_{mean} such that

$$\tau(C_{mean}, \vec{x}) = \frac{\sum_{i=1}^{N} \tau(C_i, \vec{x})}{N.}$$

Scheme **C** offers some opportunity for achieving the best performance on each input $\vec{x}$.

3.3. Parallel Speedup

Our analysis must begin with semantics, as otherwise we are subject to criticism of the "apples and oranges" type. Such criticism stems from the observation that changing the problem in order to apply a program transformation makes performance results incomparable; we are comparing unlike programs.

To an observer, the concurrent execution of the C_i must look like Scheme **B** (as discussed above); that is, that we have followed a single thread of computation, chosen arbitrarily from amongst $C_1, \ldots, C_N$. Since the $C_1, \ldots, C_N$ may update shared state described by $\vec{x}$, we solve the problem by copying state when needed and by selecting some C_i by virtue of its state changes.

By executing the C_i concurrently, we will expect the cost of execution to be

$$\tau(C_{best}, \vec{x}) + \tau(overhead)$$

where

$$\tau(C_{best}, \vec{x}) \leq \cdots \leq \tau(C_{worst}, \vec{x})$$

and *overhead* consists of operations performed to support concurrent execution which would not be necessary in the nondeterministic sequential case. It consists of (1) setting up the "Multiple Worlds", one per

alternative; (2) run-time overheads such as copying state which is updated; and (3) completion costs, such as committing the state changes made by the successful alternative and deleting its slower siblings. Parallel execution wins *iff*

$$\tau(C_{best}, \vec{x}) + \tau(overhead) < \tau(C_{mean}, \vec{x}).$$

Thus, we can calculate the performance improvement (*PI*) as:

$$PI = \frac{\tau(C_{mean}, \vec{x})}{\tau(C_{best}, \vec{x}) + \tau(overhead)}$$

essentially a ratio of execution times.

We can manipulate the simple relationships describing *PI* into forms which genuinely ease analysis.

In fact, we can analyze precisely the domains in which there is a performance improvement ($PI > 1$). Letting $R_\mu = \frac{\tau(C_{mean}, \vec{x})}{\tau(C_{best}, \vec{x})}$ and $R_o = \frac{\tau(overhead)}{\tau(C_{best}, \vec{x})}$, we can calculate *PI* as:

$$PI = \left(\frac{1}{1 + R_o}\right) \cdot R_\mu$$

This re-expression isolates the effect of the dispersion, encapsulated in R_μ , from the effect of the overhead, encapsulated in R_o. Holding one of R_μ or R_o fixed allows us to estimate the effects on *PI* caused by the other. The behaviors are illustrated in figures 3 and

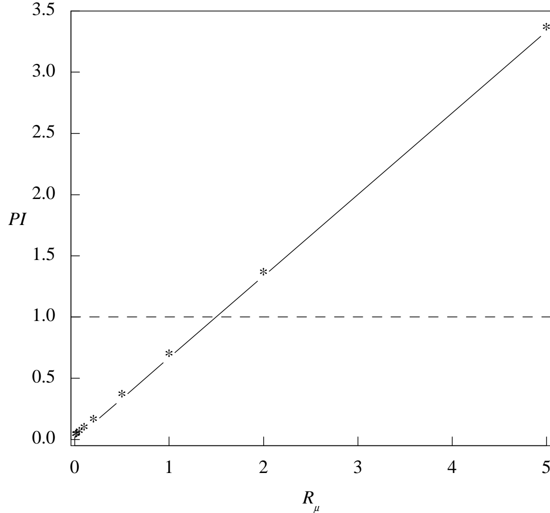


Figure 3: *PI* as a function of R_μ (R_o=0.5)

4. The relationship illustrated by the first figure is with R_o set to the constant value 0.5 (in [18] we observed a *write fraction,* which describes the fraction of memory copied by "copy-on-write" mechanisms, to be between 0.2 and 0.5. Thus 0.5 is reasonable, since the major overhead we observed was copying). R_μ is varied between 0 and 5, and the values can easily be scaled. The curve is not very interesting, as it's a direct proportion for fixed R_o; R_o determines the slope of the line, with $R_o = 0$ the best case giving a slope of 1. This tells us that the performance improvement we can expect will be proportional to the variance of $\tau(C_i, \vec{x})$, damped by whatever effect $\tau(overhead)$ exhibits. Holding R_μ fixed and varying the overhead is somewhat more interesting, as figure 4 illustrates. The *y* axis has *PI* scaled proportional to $R_\mu = exp(1.0)$, and the scales are log-log in order to view a wide range of values.

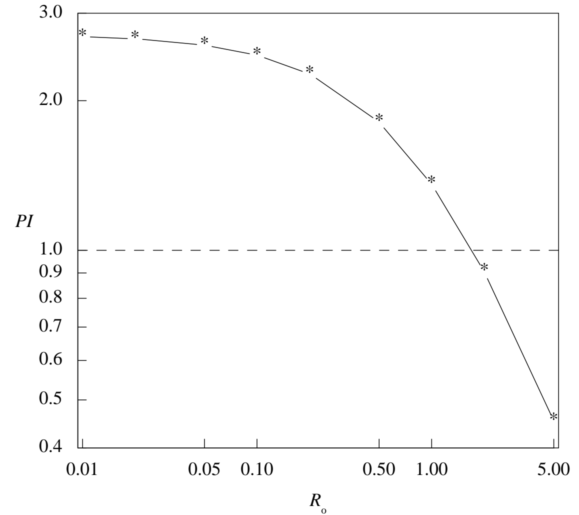


Figure 4: *PI* as a function of R_o (R_μ=*exp(1.0)*)

This tells us that varying the overhead has a significant effect on the performance improvement we achieve, when scaled against the variance in execution times. An important fact which we can deduce from this performance analysis is that with sufficient variance, and small enough overhead, N processors can exhibit superlinear speedup by parallel execution of N serial algorithms, as opposed to parallel execution of one serial algorithm which has been "parallelized".

These analyses apply to the performance on a single input; it is rather simple to extend the analysis to the entire input domain, as the metrics and independent variables in computing a performance improvement index are quite similar. One important idea which emerges when analyzing the overall performance improvement is that the different algorithms should perform well at different and unpredictable points in the input; the best case is where at each input where one or more algorithms perform badly, they have at least counterpart which performs well.

3.4. Measured Overhead

It is informative to examine measured values of possible contributors to $\tau(overhead)$. Elsewhere [18] we provide a detailed set of measurements and performance analysis 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.

There is somewhat more overhead associated with the distributed case. In Smith and Ioannidis [19] we discuss an implementation of a remote *fork()* procedure and the process migration scheme we implemented using it. An *rfork()* of a 70K process requires slightly less than a second, and network delays gave us an observed average execution time of about 1.3 seconds; we used a special-purpose remote-execution protocol which uses a network file system to reduce copying. The major cost (since we implemented *rfork()* without operating system modification) was creating a *checkpoint* of the process. The state of the process was dumped into a file in such a way that the file is executable; a bootstrapping routine restores the registers and data segments and returns control to the caller of the checkpoint routine when this file is executed. A return value is used to distinguish between return of control in the checkpoint and in the calling process. More sophisticated migration schemes, using "on-demand" state management techniques have been constructed [23]. In any case, most programs exhibit *locality of reference;* in particular symbolic computations which utilize large amounts of system resources [18]. Experiments we have done with sibling elimination schemes suggest that this can be accomplished very cheaply, e.g., on the machines we report our measurements of "copy-on-write" for, the elimination of 16 subprocesses can be accomplished in about 40 milliseconds if waiting for their termination, and 20 milliseconds if the elimination is done asynchronously.

4. Applications

What properties must we have, other than minimal implementation overhead, for the concurrent execution method we describe to be useful? We've identified the following as desirable properties:

1. There is some state shared between the alternatives which each may update. If there's no shared state, there's no point in applying "Multiple Worlds" technique, with its overhead.
2. A large portion of the shared state is read-only.
3. There are expected to be performance differences between the alternatives, due to data dependencies or use of heuristic methods.

Application areas for our design are described in the following sections.

4.1. Distributed Execution of Recovery Blocks

We have discussed the distributed execution of recovery blocks in an earlier paper [20], and will summarize here. A recovery block is composed of several alternative methods of computing a result; the goal is to emulate the behavior of "standby-spares" to tolerate faults in software. Since each alternative is guaranteed the same initial state, they can be executed concurrently. Alternatives may attempt to update shared state, e.g., database files or external variables. Our "Multiple Worlds" mechanism for preventing observation of a sibling's actions is necessary, and the "copy-on-write" memory management reduces the amount of state which must be maintained. Special modifications of "Multiple Worlds" may be necessary for fault-tolerant applications.

4.2. OR-parallelism in *Prolog*

The *Prolog* programming language is based on predicate logic, using "Horn clauses" to describe data and interrelationships. Since a *Prolog* program consists of a knowledge base combined with rules for using the knowledge and a logical inference mechanism, solutions can be modeled as an AND-OR tree of logical deductions, based on the rules as specified by clauses. The opportunities for parallelism arise because branches of this tree can be pursued in parallel, AND-parallelism where a list of clauses must all be shown true, and OR-parallelism, where at least one of a list of clauses must be shown true. OR-parallelism maps closely to our problem of attempting alternatives in parallel. The alternatives are specialized to clauses of predicate logic. Crammond [5] provides a good overview of the problems, and provides some analysis of mechanisms designed for efficient reference of

shared data, in particular updates.

Some of the solutions which have been proposed are: (1) blocking the process which updates shared state; (2) not allowing guards to update shared state; (3) sharing pointers, and hence updates, to a shared environment; (4) copying and merging. What our method does is copy, and since we choose only one alternative, no merging is necessary. (It has been argued in the *Prolog* community that sequential *Prolog* semantics be preserved, and that the major problem in OR-parallelism is multiple binding environments under this restriction. However, this argument does not admit side-effects other than variable binding, which seems short-sighted in terms of real applications. The sort of *committed-choice nondeterminism* we advocate here is popular in another segment of the *Prolog* community addressing OR-parallelism.) Since there are no extra (beyond whatever is required for sequential execution) pointer chains to traverse on variable references, memory access is fast. Use of the method requires changing the *Prolog* interpreter to detect and exploit OR-parallelism. How aggressively available parallelism is exploited 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.

4.3. Numerical Applications

Polyalgorithms [15] have been suggested as a method for encapsulating a numerical analyst's knowledge into a system for solving numerical problems. The basic idea is that several methods are combined along with information about the circumstances under which a method is likely to be successful. As different methods are tried and fail, information about the problem is built up until either there are no successful solutions, or a solution method succeeds (for example, discovering multiple zeros in a failing root-finder may be useful to the next solution method).

"Multiple Worlds" could be used by creating artificial "alternatives" with the available solution methods. Each "alternative" trys a different solution method "first," to create alternative versions of the polyalgorithm. "Fastest first" scheduling could improve the response time properties of a system such as NAPSS [16], especially since the performance of the system was perceived to be a problem [17].

Another possibility is the exploitation of a degree of freedom in a solution method, as in choosing several values for an ostensibly random choice. Using polar coordinates, the angle of the starting value is a random choice in the complex version of the Jenkins-Traub [11] polynomial zero finder. In practice, several angles are tried, based on numerical experience. A parallel version of this algorithm was created [21] by making several choices for the staring value and executing them in parallel. A two processor Ardent Titan produced the results of Table I.

procs	max	min	avg	fails	par
1	4.01	4.01	4.01	0	4.37
2	4.49	4.07	4.28	0	4.25
3	4.45	2.03	3.50	0	4.74
4	4.48	1.37	3.31	0	5.19
5	4.27	2.36	3.35	2	8.61
6	4.50	2.02	3.65	0	7.03

Table I: Parallel Rootfinder

All times are in seconds. The first column, labeled **procs**, indicates the number of processes applied to the problem. Ideally, there would be one processor for each process, but only 2 processors were available. 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; the execution time overhead of creating two processes and running them concurrently can be computed as 4.25-4.07 sec., or about 0.18 sec. But the average time was 4.28 sec, so even with the additional overhead, the parallel execution finished first. The performance in the 4 process case would be much better if there had been more than two processors available. Performance on processors with higher degrees of parallelism is under investigation.

5. Related Work

The IBM 360 Model 91 [1] 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 observable side effects at synchronization.

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* [12, 22] 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 [4] 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. Goldberg [8] has also discussed process replication, with a focus more on performance than fault tolerance. 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, as was illustrated by Jefferson's [10] use of a specialized buffering process called *stdout.* Transparent replication can easily be combined with the use of parallel execution of several alternatives for increases in performance, reliability, or both.

Wilson [24] proposes "Alternate Universes", which he developed independently of our "Multiple Worlds" scheme. The major difference we see is that Wilson's approach is value-based (and so might be incorporated in a language in order to exploit fine-grained parallelism) while our scheme is page-based and hence suitable for larger-grained parallelism; "Multiple Worlds" interaction with the memory management portion of an operating system trades a higher startup cost against cheaper referencing from that point on, at least on existing general purpose computers.

6. Conclusions

When (1) alternatives require a significant amount of computation time; (2) each alternative changes a small amount of the state of the calling process, thus reducing the penalty of $\tau(overhead)$; and (3) there is a significant variance in the execution times of the alternatives, "Multiple Worlds" can be applied. The new performance analysis of section 3 discusses the relationship between the factors in a speedup, and applies the analysis across a domain, rather than locally [20].

The "Multiple Worlds" scheme ensures that any performance improvement is achieved in a manner which is transparent to the application programmer. Several instances of application domains with appropriate characteristics were discussed, and encouraging initial results from a multiprocessor execution were presented.

7. Notes and Acknowledgments

Robert Strom, Calton Pu, Yechiam Yemini, Steve Feiner, David Farber, Sal Stolfo, and Andy Lowry have refined our ideas, through observations, suggestions, and insightful criticism. Suggestions from anonymous referees for the 1989 International Conference on Parallel Processing led to several important revisions.

UNIX and WE 32101 are registered trademarks, and 3B2 is a trademark of AT&T; HP-UX, HP9000, and HP are trademarks of the Hewlett-Packard Corporation.

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

8. References

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, pp. 8-24 (January 1967).

[2] AT&T, *WE 32101 Memory Management Unit Information Manual*, Call 1-800-432-6600; Select

Code 307-731, November 1986.

[3] 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).

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

[5] J. Crammond, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages," *IEEE Transactions on Computers* **C-34**(10), pp. 911-917 (October 1985).

[6] Bryce DeWitt and R. Neill Graham, *The Many Worlds Interpretation of Quantum Mechanics*, Princeton University Press, 1973.

[7] 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).

[8] Arthur P. Goldberg and David R. Jefferson, "Transparent Process Cloning: A Tool for Load Management of Distributed Programs," in *Proceedings, International Conference on Parallel Processing* (1987), pp. 728-734.

[9] James P. Hogan, *Thrice upon a Time*, Ballantine, 1980.

[10] 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).

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

[12] H. T. Kung and John T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems* **6**(2), pp. 213-226 (June, 1981).

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

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

[15] J. R. Rice, "On the Construction of Polyalgorithms for Automatic Numerical Analysis," in *Interactive Systems for Experimental Applied Mathematics*, ed. J. Reinfelds (1968), pp. 301-313.

[16] John R. Rice, "NAPSS-like systems: Problems and Prospects," in *Proceedings, National Computer Conference* (1973), pp. 43-47.

[17] John R. Rice, *Private Communication on NAPSS*, October, 1988.

[18] 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: The Journal of the USENIX Association* **1**(3), pp. 255-278, University of California Press (1988).

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

[20] 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), pp. 44-52.

[21] Jonathan M. Smith, "Concurrent Execution of Mutually Exclusive Alternatives," Ph.D. Thesis, Columbia University Computer Science Department (May, 1989). also available from UMI

[22] 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

[23] 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.

[24] Paul R. Wilson, *Two Comprehensive Virtual Copy Mechanisms,* University of Illinois at Chicago, Electrical Engineering and Computer Science, Chicago, Illinois (1988). M.S. Thesis