# Transparent Concurrent Execution of Mutually Exclusive Alternatives

*Jonathan M. Smith*

*Gerald Q. Maguire, Jr.*

Computer Science Department, Columbia University, New York, NY 10027
Technical Report Number CUCS-387-88†

*ABSTRACT*

We examine the task of concurrently computing alternative solutions to a problem. We restrict our interest to the case where only one of the solutions is needed; in this case we need some rule for selecting between the solutions. We use ''fastest first'', where the first successful alternative is selected. For problems where the required execution time is unpredictable, such as database queries, this method can show substantial execution time performance increases. These increases are dependent on the mean execution time of the alternatives, the fastest execution time, and the overhead involved in concurrent computation.

Among the problems with exploring multiple alternatives in parallel are side-effects and 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.

In order to test the utility of the design, we show how it can be applied to two application areas, distributed execution of recovery blocks and OR-parallelism in Prolog.

**Topic Designators:** Distributed and Parallel Algorithms, Experimental Distributed Systems, Modeling and Performance Evaluation, Languages

**Additional Keywords:** Transparency, Speculative Computation, Memory Management

† A version of this paper will appear in the Proceedings of the 9th International Conference on Distributed Computing Systems.

# Transparent Concurrent Execution of Mutually Exclusive Alternatives

*Jonathan M. Smith*

*Gerald Q. Maguire, Jr.*

Computer Science Department, Columbia University, New York, NY 10027

Technical Report Number CUCS-387-88†

## 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. 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*, which is a measure of the amount of useful work accomplished per unit time. Given this bias, we may risk wasted work in *speculative* computation [Burton1985a], 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.

Once the model is defined, and the semantics thus fixed, we can apply semantics-preserving transformations in order to increase performance or achieve other goals. A successful transformation, then, has two requirements. First, it must correctly preserve the semantics. Second, it must achieve the goal set for it, e.g., a performance increase.

We present (1) a model for selection of alternatives in a sequential setting, (2) a transformation which allows alternatives to execute concurrently, (3) a description of the semantics-preservation mechanism, and (4) parameterization of where the performance improvements can be expected. Additionally, we show example application areas for our method.

## 2. Sequential Model

Consider the situation where 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. An ALGOL-like language construct embodying this situation:

```
ALTBEGIN
    ENSURE guard1 WITH method1 OR
    ENSURE guard2 WITH method2 OR
                  .
                  .
                  .
    ENSURE guardn WITH methodn OR
    FAIL    /* no method succeeded */
END
```

**Figure 1:** Alternative Block

What we want is for at most one of the methods to be applied to our problem, or for whatever conditions constitute failure to be indicated. Each method, 1..*n*, has associated with it a *guard* condition, which it must satisfy in order to be considered successful. A method is called an *alternative*. When the alternatives are composed into a block, as illustrated in figure 1, the meaning is that one of the alternatives (including failure) are selected non-deterministically. The non-determinism in selection is necessary for higher-performance computing. The selection is non-deterministic and *unfair*, in that the selection of alternates is not equiprobable, and should not be; it's clear that the alternative of failure should be given as low

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

a probability of success as is possible, noting that when all the alternatives fail its conditional probability must be 1. The semantics of the construct behave similarly to Dijkstra's [Dijkstra1976a] guarded commands, in the special case where the same guard is used for all the statements. In an implementation setting, the construct resembles the Ada `select` with guarded alternatives; the selection of open (i.e., have satisfied the guard) alternatives is arbitrary.

## 3. Parallel Execution

### 3.1. System Model

A *process* 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, *source* and *sink*. The division is made on the basis of idempotence; operations on *sink* devices can be retried without the effects being visible, while operations on *sources* cannot be retried. For definiteness, 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 external to 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 [Organick1972a]. 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 [Sandberg1985a] can be utilized to hide the network through the page management abstraction.

### 3.2. Process Management

Two primitives encapsulate the entire semantics of the process management component. The process management component is concerned with the mutually oblivious alternatives. To spawn the alternatives, the parent uses `alt_spawn( n )`, which returns numbers from 1 to n in the alternates 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:
      alt_wait( TIMEOUT );
      fail(); /* if returned */

   case 1:
      /* First alternate */
            .
            .


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

The purpose of `alt_wait()` is manifold; the essence is establishing a single path through the tree of possible computations which is reflected in the execution history of the running process. `Alt_wait()` takes a TIMEOUT value as an argument; the point is that this value should be chosen such that if TIMEOUT time units have elapsed, it is highly probable that none 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. The point of passing such a timeout value will be seen shortly.

When a spawned alternate calls `alt_wait()` at the termination of its computation, a rendezvous between the `alt_wait()`ing parent and the child is effected. The behavior is much like that of the UNIX `exec()` system call, where the new data and executable code are read in from a named file. In the case of `alt_wait()`, the parent process absorbs the state changes made by its child by atomically replacing its page pointer with that of the child. 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 execution of the program segment in figure 1 shown in figure 2:

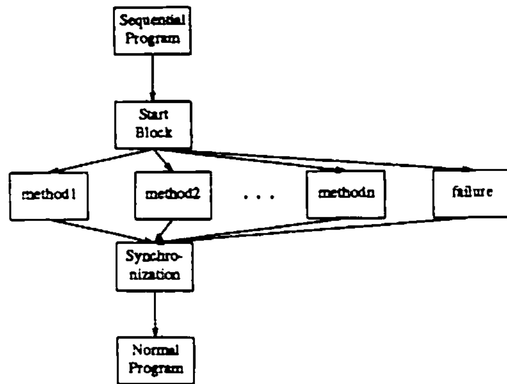


Figure 2: Concurrent Execution of Alternates

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 GUARD can be executed before spawning the alternative, in the child process, at the synchronization point, or at any combination of these places, for redundancy. We currently expect the child process to execute it, thus speeding up spawning and synchronization.

### 3.2.1. Synchronization

It is at the synchronization point that the data for sibling elimination are available; all processes which assumed that the successful child had failed must be deleted, as they have made an assumption we know to be false. 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 process is to be terminated. The deletion can be accomplished synchronously (where the other alternates 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); we suspect that asynchronous elimination will give better execution-time performance, once again at the expense of resource utilization measures such as throughput.

Now, communications problems or system failures may prevent this information from reaching the scheduling component of a remote system, yet we must still preserve the ''at most one'' semantics of our design. The backup in this case is that the synchronization action is designed so that it can be accomplished at most once; that is, if the remote system attempts synchronization for the alternative it is executing, it is informed that it is ''too late'' for the synchronization, and it should terminate itself. In applications where this might create a single point of failure, the synchronization is set up as a majority consensus [Thomas1979a] decision across several nodes. The engineering tradeoff here is between performance and reliability; the additional communication and protocol of multiple-node synchronization is the price paid for increased robustness of the synchronization.

### 3.3. 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 then the predicates described by Eswaran, *et al.*[Eswaran1976a] 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[1]. The state management strategy is ''copy-on-write'' [Bobrow1972a] with page map inheritance from the parent, thus it is easily implemented within the context of a system which provides such features, e.g., Mach [Young1987a], and benefits from existing hardware support, e.g., for the WE® 32101 MMU [AT&T1986a]. 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.

### 3.4. Interprocess Communication

### 3.4.1. 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.

[1] Thus, so-called ''sibling rivalry'' is taken to its extreme in this design! The failure alternative assumes that none of the siblings will complete.

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.

### 3.4.2. Multiple Worlds

An idea from science fiction, inspired by Dewitt's [DeWitt1973a] 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 ( 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., $S \subseteq R$ ), the message is immediately accepted. If the receiver's predicates conflict ($p \in S$ and $\neg p \in R$), the message is ignored, and if the receiver must make further assumptions to accept the message ( $p \in S$ and $p \notin R$), two copies of the receiver are created. One of these copies is created with the predicates set to the previous values in conjunction with `complete( S )`[2]; the other is set up with its predicates as before, except that `complete( S )` is negated.[3] 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*.

This behavior is similar to that required of *transactions*. Transactions [Gray1978a] are a structuring concept for operations; transactions are required to be atomic with respect to any observer.

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

[3] 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.

## 4. 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 alternate is use of available hardware.

Note that the action of continuing execution of the successful alternative and the process of sibling elimination can take place *asynchronously*. The effects of various overheads and system parameters are analyzed in the next section.

### 4.1. Overhead

In order to understand the overhead implied by the method, we should compare a sequential execution of the construct, in the best case, where the fastest alternative is selected. There are penalties we are paying for parallel execution of all alternatives versus sequential execution of the alternative which will be selected in any case. These are

1. Memory Copying. In the distributed case we must actually copy state for a remote child so that it 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. 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 asynchronous, and naturally parallel, but the instructions to terminate the alternates must still be issued, and they increase with the number of alternates.
3. Effect on throughput, or wasted work. As our bias has been towards execution time as a performance goal, we were willing to trade away throughput. Users may want to know what the tradeoffs are here, so the effect on system throughput should be analyzed.

### 4.2. Analytic Description

Assume that we have $N$ alternative methods of performing a *computation*. A *computation* is a transformation from an input set (or Domain) to an output set (or Range); these sets consist of *state vectors*, intended to describe the relevant state of the world, i.e., the machine state. For Domain **D** and Range **R**, $\vec{x} \in \mathbf{D}$ is transformed via the computation into some $\vec{y} \in \mathbf{R}$, thus we could write $\vec{y} = C(\vec{x})$. There may be several such $C$ which we classify as interesting (transformations of $C$ which add or remove useless operations are infinitely numerous, but not interesting. *Algorithmic* differences or significant

differences in implementation technique are interesting.). Assume that the $N$ alternatives postulated earlier are $N$ such interesting $C$s, and that they will be applied to some $\vec{x} \in \mathbf{D}$. Each $C$ consists of some series of *steps*, where $\vec{x}$ is transformed into $\vec{x}'$, $\cdots$ until $\vec{y}$ is achieved. Each step requires some amount of clock time, $\tau$, to complete; for $C(\vec{x})$, $\tau(C,\vec{x})$ is the sum of these times. $\tau$, the *execution time*, gives us a way of comparing the performance of two computational methods on the same input, say $\vec{x}$.

There are many practical situations in which we want to minimize the computation time required for the transformation of $\vec{x}$ to $\vec{y}$. We will denote the $N$ alternatives as $C_1, \ldots, C_N$. Since our goal is minimizing execution time, let us consider some possible relations between the $C_i$ on elements of $\mathbf{D}$.

1. $\tau(C_i,\vec{x}) \le \tau(C_j,\vec{x})$ for every $\vec{x} \in \mathbf{D}$ which interests us. It's clear that we should use $C_i$ and discard $C_j$ for every $i$ and $j$ for which this holds.

2. $\tau(C_i,\vec{x}) \le \tau(C_j,\vec{x})$ for some $\vec{x}$ which interest us, and we can accurately predict for which $\vec{x}$ this relation holds. In this case, we can construct a synthetic computation, $C_{N+1}$, which selects $C_i$ when this holds. To anchor the relation with an example, consider the case of two list-sorting algorithms, `Q` and `I`. `Q` is faster than `I` when the number of elements to be sorted is greater than 10. Thus, using this knowledge, we can construct a synthetic sorting routine as follows:

```
sort( list, size ) :=
    if( size > 10 )
        Q( list, size )
    else
        I( list, size ).
```

   The synthetic routine partitions the input domain by performance, and thus achieves performance superior to either `Q` or `I`. The tough point here is the partitioning; it's rarely as simple to delimit performance boundaries as "`size < 10`". If the input set can be partitioned, but only at significant computational cost, the desired property of the synthetic routine, that $\tau(C_{N+1},\vec{x}) \le \tau(C_i,\vec{x}) \forall i$, for all $\vec{x}$ of interest, may be achievable with the following technique.

   If all interesting $\vec{x}$ are known in advance, we can associate one of the $C_i$ with each $\vec{x}$ in a precomputed table. Then, $\tau(C_{N+1},\vec{x})$ can be calculated by adding the cost of a table lookup to the cost of executing the table element on $\vec{x}$.

3. $\tau(C_i,\vec{x}) \le \tau(C_j,\vec{x})$ for some $\vec{x}$ which interest us, but while interesting, the $\vec{x}$ cannot easily be related to $\tau(C_i,\vec{x})$. Essentially, this means that the table lookup technique cannot be used, because we cannot reasonably precompute the values of $\tau(C_i,\vec{x})$. This might be due to the nature of the input set, e.g., infinite size. For example, a naive *quicksort* is not stable, and where the list is *ordered* the sort is slow. In these cases, a stable sort with good performance, e.g., *heapsort*, may be preferable. However, it's clear that storing a lookup table of of all "interesting" lists is infeasible, and pretesting for the "ordered" property is potentially quite expensive. Another problem is that $\tau(C_i,\vec{x})$ may vary due to the execution environment (which may or may not be described by $\vec{x}$; it probably should be, for completeness), e.g., processor type, multiprocessing workload, or interactions with other computations. In these cases, where performance on the $\vec{x} \in \mathbf{D}$ is unpredictable, we might try other schemes:

   **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 $\vec{y}$ is selected. The other $C_i$ are irrelevant and can be terminated. There is, however, overhead in setup and synchronization (selection) which cannot be ignored.

   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., $\frac{\sum_{i=1}^{N} \tau(C_i,\vec{x})}{N}$.[4] Scheme **C.** offers some opportunity for achieving the best performance on each input $\vec{x}$. We will try to characterize this opportunity. Note that there are two possibilities for concurrent execution, *real* and *virtual*. Real concurrency means that the evaluation of $C_i(\vec{x})$ is taking place simultaneously with that of $C_j(\vec{x})$; virtual means that there is some sharing of hardware, for example through multiprocessing.

### 4.3. 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

[4] It is interesting to note, as well, that failures or infinite loops will frustrate this method.

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. Thus, since the observer sees non-deterministic selection of one of the alternatives, we must compare concurrent execution to sequentially performing one of the $C_i$, chosen arbitrarily (we'll assume randomness). Since, as stated previously, execution time is our figure of merit, we'll analyze with that intent, ignoring measures such as throughput. Arbitrary selection can be done by a call to a random number generator, which costs nothing for purposes of our analysis. The execution of the selected alternative costs $\tau(C_i,\vec{x})$ for the $\vec{x}$ under study. Thus, we can expect the mean cost to be $\frac{\sum_{i=1}^{N}\tau(C_i,\vec{x})}{N}$, the average of the $C_i$s times when applied to $\vec{x}$.

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

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

where

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

and *overhead* is quite complex. *Overhead* consists of operations performed to support concurrent execution which would not be necessary in the nondeterministic sequential case. It consists of the following components:

setup: Instead of simply calling $C_i$, we must now spend cycles creating execution environments for $C_1, \ldots, C_N$; for example, setting up process table entries and page map tables.

runtime: This consists of copying memory areas which are shared between the $C_1, \ldots, C_N$ when updates are attempted. This performance is strongly influenced by *locality of reference*. Additionally, if $C_{best}$ is sharing resources, e.g., CPU time, with some $C_i$, $i \ne best$, then for all such $C_i$, $C_i$'s runtime must be added to the runtime overhead of $C_{best}$, as cycles spent processing $C_i$ are not spent processing $C_{best}$.

selection: This is the cost involved in selecting $C_{best}$, e.g., deleting $C_i$ *such that* $i \ne best$, cleaning up system state, such as actually performing the updates made by $C_{best}$, e.g., writing checks or bottling beer.

Thus, for a given $C_1, \ldots, C_N$ and $\vec{x}$,

$$\tau(overhead) = \tau(setup(C_1 \cdots C_N,\vec{x})) + \tau(runtime(C_{best},\vec{x})) + \tau(selection(C_{best},C_1, \ldots, C_N,\vec{x})),$$

and the parallel execution wins *iff*

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

For notational convenience, define $C_{mean}$ such that

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

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. For illustration, consider a case where $N$=3, on input $\vec{x}$. Thus, we have three methods $C_1$, $C_2$, and $C_3$. Let $\tau(overhead)$ be 5. Some possible relations are tabulated:

| | $\tau(C_1,\vec{x})$ | $\tau(C_2,\vec{x})$ | $\tau(C_3,\vec{x})$ | *PI* |
|---|---|---|---|---|
| (1) | 10 | 20 | 30 | 1.33 |
| (2) | 1 | 19 | 106 | 7.0 |
| (3) | 20 | 20 | 20 | 0.8 |
| (4) | 1 | 2 | 3 | 0.33 |
| (5) | 115 | 120 | 125 | 1.0 |
| (6) | 100 | 200 | 300 | 1.9 |

What can we infer from the examples? (3) indicates, along with (5), that the size of the differences matters. (4) shows that the relative magnitudes of the execution times and the overhead matters. (6) shows that the effects of the overhead (under our assumptions) diminish with increasing relative execution time. (2) illustrates a good situation, where the difference

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

is very large. This magnitude of difference is well-encapsulated by such a statistical measure of dispersion (letting values of $\tau$ serve as the random variable) as the *variance*.

## 4.4. Measured Overhead

It is informative to examine measured values of possible contributors to $\tau(overhead)$. In another report [Smith1988a] 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 [Smith1989a] 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[5] in its entirety. More sophisticated migration schemes, using "on-demand" state management techniques have been constructed [Theimer1985a]. In any case, most programs exhibit *locality of reference;* in particular symbolic computations which utilize large amounts of system resources [Smith1988a].

## 5. 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. A large portion of the shared state is read-only.
2. There is some state shared between the alternatives which each may update.
3. There are expected to be performance differences between the alternatives, due to unknown data characteristics or use of heuristic methods.

Two application areas for our design are described in the following sections.

### 5.1. Distributed Execution of Recovery Blocks

The Recovery Block [Horning1974a] is a method for writing software which is tolerant of mistakes in its own logic, from which failures can arise. The idea is quite simple. It is assumed that the software in question has been written to some specification. Several alternative versions of the software are written, according to the specification. A boolean "acceptance test", which checks the results of the software is developed along with the software, using the specification. The acceptance test, which either succeeds or fails, will be refined once some experience with the software is developed.

The alternatives and the acceptance test are gathered into an ALGOL-like block construct, where the alternatives are typically ordered on the basis of observed or estimated characteristics such as reliability and execution speed.

When the acceptance test succeeds, the results (including all state changes) of the alternative which passed the test are made available. When the acceptance test fails, the state of the program is "rolled back" to the state the program had before the block was entered, and the next alternative is tried. If the last alternative in the sequence results in a failed acceptance test, the block as a whole fails.

#### 5.1.1. Sequential Model

The recovery block is somewhat different in behavior than the "Alternative Block" we proposed as a sequential model in Section 2. First, rather than having one guard per body, the Recovery Block possesses one guard to which all the alternatives are passed. Second, the guard is applied *after* the body is executed, rather than before. However, neither of these are problems for our design, as (1) the computation can be viewed as part of the guard, with the body consisting solely of updates to external variables, or (2) the blocks can be viewed as self-checking entities where the guard is always enabled for scheduling of the computation, which may fail due to self-checks.

The changes to the program's state space are equivalent to some execution which selected exactly one of the alternatives (or failure) at each Recovery Block. Thus, this is exactly the nondeterministic selection which we chose for our model, and it should be all that a *post facto* examiner of the program state can deduce.

[5] We do this by dumping the state of the process 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.

### 5.1.2. Concurrent Execution

Since Recovery Block alternates may attempt to update shared state, e.g., database files or external variables, our 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. One special problem which arises with the parallel execution of Recovery Block alternates[6] is the fact that the method is designed to cope with failures, so that we must do more work in order not to add new failure modes. Two issues in particular are important. First, we may copy all of the state rather than copying as necessary, in order that the state not become inaccessible and so cause a failure. Second, the synchronization must not introduce a single point of failure. This is remedied by the use of majority consensus, as discussed above, to achieve a fault-tolerant 0-1 semaphore for use in synchronization.

## 5.2. OR-parallelism in *Prolog*

The *Prolog* [Clocksin1984a] programming language is based on predicate logic, using "Horn clauses" [Rich1983a] to describe data and interrelationships. Many normal operations are subsumed by the unification algorithm by which *Prolog* attempts to satisfy predicates; variables are bound during the unification process to values which caused the predicates to become true. Thus `equal(X,elrod)` will cause the variable `X` to take on the value `elrod`, as this binding is the only one which allows the predicate `equal()` to be satisfied.

Progress is achieved with a goal-oriented predicate-satisfaction algorithm; a database of predicate values and rules is used to construct a set of dependency relations; top-level goals are decomposed into sub-goals using the relations between the rules, objects, and predicates. For example, testing equality of lists implies that their elements are equal; testing element-wise equality may then give a list of sub-goals. This gives rise to a possibility for parallel execution, however the granularity of such parallelism seems inappropriate. More appropriate is rule-level parallelism, which is centered on two types, AND-parallelism and OR-parallelism. The idea with AND-parallelism is that if we have a situation where goals A and B must be satisfied, we can pursue the satisfaction of A and B in parallel. The situation is similar for OR-parallelism; this is more interesting to us, since it maps closely to our problem of attempting alternatives in parallel. The alternatives here are specialized to predicates. Crammond [Crammond1985a] provides a good overview of the problems, and provides some analysis of mechanisms designed for efficient reference of shared data, in particular the update of shared data.

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. 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.

## 6. Related Work

Exploring alternatives in parallel is far from a new idea; hardware engineers looked to it as a way of maintaining pipeline utilization in some high-speed computers, most notably the IBM 360 Model 91 [Anderson1967a]. 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 go further.

Version control systems such as SCCS [Rochkind1975a] use the idea of deltas to store multiple versions of data. More related to our *predicates* is the idea used in the PEDIT [Kruskal1984a] parametric line editor. Associated with each line of text is a set of *parameters*. These parameters are state variables, e.g. `SYSTEM=UNIX`, `VERSION=SysV`, et cetera. The line is selected for display if the mask set in the view of the file matches the settings of the state variables; thus, the viewer of a source program in a particular environment might see the source without the obscuring effect of various conditional compilation directives. Each setting of the state variables gives a distinct *version*, but in practice most of the text is shared between the versions.

Our method uses 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. These predicated processes are similar to the *possibilities* and *dependencies* discussed by Reed [Reed1978a] in his thesis; however, his

[6] See the work of Kim [Kim1984a] and Welch [Welch1983a] for a discussion of the distributed execution of recovery blocks. They describe the performance increases possible using concurrent execution; they used two-alternate recovery blocks on a bus-connected shared memory multiprocessor for their experiments.

NAMOS system was somewhat further from realization than the methods described here.

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. It could also be viewed as a set of "competing" transactions, at most one of which will take effect.

One significant feature of our use of *predicates* there is little *waiting* as possible in the system; each process which could execute under any set of assumptions makes that set of assumptions, until some conflict with the correctness policies results. In other settings, such methods are called *optimistic* [Kung1981a, Strom1987a] 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*. In fact, Strom and Yemini's [Strom1985a] *dependency vectors* behave much like our predicates.

Distribution of computation across several nodes offers attractive possibilities for both reliability and performance. Cooper [Cooper1985a] discusses the use of replicated distributed programs in order to take advantage of this potential. Cooper's CIRCUS [Cooper1984a] system transparently replicates computations across several nodes in order to increase reliability. Goldberg [Goldberg1987a] 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 count 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 best sort of situation for our approach is one where:

- Alternatives require a significant amount of computation time, as encapsulated in $\tau(C_{mean}, \vec{x})$.
- Each alternative changes a small amount of the state of the calling process, thus reducing the penalty of $\tau(overhead)$.
- There is enough difference between the execution times of the alternatives that choosing the fastest and killing the others is worth the overhead of spawning the copies and deleting the slower siblings. This may also be true in real-time systems, where the sibling elimination can be carried out asynchronously with respect to result delivery.

It appears that parallel implementation of logic programming languages provides such an environment, because the computation is data-driven, and thus the execution time and control flow can vary greatly with the input. The way in which unification operates (as a "sophisticated pattern matcher") leads to an overwhelming preponderance of read references made to page-managed memory; while a high *percentage* of references are writes, these are mainly to the stack, and thus locality should be quite high.

Distributed execution of recovery block alternates uses the "fastest-first" behavior in an attempt to find a rapid failure-free path through the computation.

## 8. Notes and Acknowledgments

Robert Strom has been extremely helpful in our gaining an understanding of the problems, approaches, and trade-offs; he has inspired many of the ideas we've presented here. Discussions with Calton Pu, Yechiam Yemini, Steve Feiner and David Farber have contributed to what we present in this report. Sal Stolfo pointed out *Prolog* OR-parallelism as an application, and Andy Lowry pointed out a flaw in an earlier presentation of the predicate scheme for IPC.

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.

## 9. References

[Anderson1967a] 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).

[AT&T1986a] AT&T, *WE 32101 Memory Management Unit Information Manual*, Call 1-800-432-6600; Select Code 307-731, November 1986.

[Bobrow1972a] 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).

[Burton1985a] F. W. Burton, "Speculative Computation, Parallelism, and Functional Programming," *IEEE*

*Transactions on Computers* C-34(12), pp. 1190-1193 (December 1985).

[Clocksin1984a] W. F. Clocksin and C. S. Mellish, *Programming in Prolog (2nd Edition)*, Springer-Verlag (1984).

[Cooper1984a] Eric Charles Cooper, "Circus: A replicated procedure call facility," in *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems* (October 1984), pp. 11-24.

[Cooper1985a] Eric Charles Cooper, "Replicated Distributed Programs," Ph.D. Thesis, University of California, Berkeley (1985).

[Crammond1985a] 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).

[DeWitt1973a] Bryce DeWitt and R. Neill Graham, *The Many Worlds Interpretation of Quantum Mechanics*, Princeton University Press, 1973.

[Dijkstra1976a] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

[Eswaran1976a] 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).

[Goldberg1987a] 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.

[Gray1978a] J. N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, ed. G. Seegmueller (1978), pp. 393-481. Springer

[Horning1974a] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery.," in *Proceedings, Conference on Operating Systems: Theoretical and Practical Aspects* (April 1974), pp. 177-193.

[Kim1984a] K.H. Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults," in *IEEE Fourth International Conference on Distributed Computing Systems* (1984), pp. 526-532.

[Kruskal1984a] V. Kruskal, "Managing Multi-version Programs with an Editor," *IBM Journal of Research and Development* 28(1), pp. 74-81 (January, 1984).

[Kung1981a] 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).

[Organick1972a] Elliott I. Organick, *The Multics System*, Massachusetts Institute of Technology Press (1972).

[Reed1978a] David P. Reed, "Naming and Synchronization in a Decentralized Computer System," Technical Report 205 (Ph.D. Thesis) (September, 1978). MIT LCS

[Rich1983a] Elaine Rich, *Artificial Intelligence*, McGraw-Hill (1983).

[Rochkind1975a] M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering* SE-1, pp. 364-370 (1975).

[Sandberg1985a] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon, "The Design and Implementation of the Sun Network File System," in *USENIX Proceedings* (June 1985), pp. 119-130.

[Smith1988a] 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).

[Smith1989a] Jonathan M. Smith and John Ioannidis, "Implementing remote *fork()* with checkpoint/restart," *IEEE Technical Committee on Operating Systems Newletter*, Also available as Columbia University Computer Science Department Technical Report CUCS-365-88 (supersedes CUCS-275-87) (February, 1989).

[Strom1985a] R. E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems* 3(3), pp. 204-226 (August 1985).

[Strom1987a] 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

[Theimer1985a] 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.

[Thomas1979a] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems* 4(2), pp. 180-209 (June 1979).

[Welch1983a] H.O. Welch, "Distributed Recovery Block Performance in a Real-Time Control Loop," in *Proceedings, IEEE Real-Time Systems Symposium* (1983), pp. 268-276.

[Young1987a] 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).