Speculative Nondeterminism

Abstract

We propose a new programmable concurrency control framework called speculative nondeterminism for real time, "open" and distributed agents. In this framework, dynamic and concurrent agent programs affect each other through a centralized store which represents shared resources. Our framework has novel programming constructs which allows an agent to speculate against the future by multiple exclusive choices that encode different strategies to achieve its goals. The agent can make multiple choices which execute together but in isolation, thus, there is a combinatorial effect through the joint interaction. Unlike other concurrency frameworks, speculation models many potential outcomes and thus improves the chances of an agent reaching a desired outcome. Each possibility is represented by a "virtual world." Agents operate in the virtual worlds created by the system, the framework provides mechanisms for agents which have reached their objective but yet are represented by multiple possible worlds to cleanly exit. We also provide mechanisms to reduce the possibilities where an agent can commit to certainly possibilities and give up on others, but this is rather different from conventional committed choice. This paper presents the design, operational semantics of speculative nondeterminism, small but practical examples illustrating the use of the language and experimental results of a prototype.

Keywords speculation, concurrency, nondeterminism, agents

1. Introduction

"Over and above the actual world there are an indefinite multiplicity of merely possible worlds... The actual world is a possible world. The other possible worlds, the merely possible worlds, are ways that the actual world might have been." [1] – D.M. Armstrong

Concurrent programs often show non-determinism. Furthermore, there are many important situations where non-deterministic choices are needed in algorithms. Programming constructs or support for non-deterministic choices fall into two classes: (i) *don't-know non-determinism*; and (ii) *don't-care non-determinism*. Don't-care non-determinism is exemplified by *committed choice* such as Dijkstra's Guards [7] and in concurrent logic or constraint programming languages [24]. In don't-care non-determinism, although there is a choice, the selection of which choice to execute is arbitrary (though there may be guards to restrict availability of choices). Thus, it is the responsibility of the programmer to deal with the consequences of the selection. Don't know non-determinsm, on the other hand, recognizes that it might not be possible to make the choice up front. In a sequential language, *backtracking* is a typical approach. This is typified by logic programming languages such as Prolog which searche the combinatorial space of all possible choices to find which choices give the desired solution. There are also languages which provide parallel search, i.e. some logic programming languages give OR parallelism, but like backtracking, it is still a search process. When one wants to deal with problems where there is both don't know non-determinism and concurrency, the situation can become more complex, which is what we deal with in this paper.

[Copyright notice will appear here once 'preprint' option is removed.]

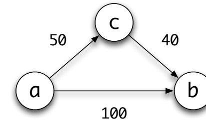


Figure 1. Flight Map and Cost Between 3 Cities

The following airline ticketing scenario exemplifies our setting and the kind of problems where we have evolving don't know non-determinism. Suppose a travel company S sells airline tickets to its customers through an online website, and now a traveler T wants to go from a to b. Such tickets are instant purchase only with a small delay to verify and complete the transaction. Figure 1 shows two possible routes, he can fly from a to b (i.e. $a \rightarrow b$), or from a to an intermediate city c, and then to b (i.e. $a \rightarrow c \rightarrow b$). Suppose T has only \$100, and tickets $a \rightarrow b$, $a \rightarrow c$ and $c \rightarrow b$ cost \$100, \$50 and \$40, respectively. Suppose S obtains bulk tickets from airlines and puts them up for sale on the website dynamically, according to its own business logic. Customers log on and purchase tickets independently. So from T's perspective, tickets come and go in *real time*. To secure his trip, T has only two possible strategies:

1. Buy the ticket which comes first, hence, commiting to the corresponding route.
2. Try one of the route first, and if he can't get the tickets to complete the route, try the other route.

These two strategies are analogous to committed choice and backtracking (although backtracking is extremely time limited), respectively. In the first strategy, when S sells $a \rightarrow c$ first and never sells $c \rightarrow b$, T gets stuck after buying $a \rightarrow c$, even if S puts $a \rightarrow b$ for sell later, as T doesn't have enough money to buy both routes (only \$50 left after buying $a \rightarrow c$). In the second strategy of T, there is no explicit cancellation but there is a very short time window between selection and actual purchase, so it is possible to abort within a short time. However, it is unlikely, that the second strategy would lead to a different situation than the first.

1.1 Speculation

The previous example shows that, in many situations, there is competition for limited resources (e.g. cash and tickets). The traveler can only attempt one of the two routes at a time. The drawback is of the previous strategies (committed choice or backtracking) are sub-

```
(wait for a → b
 buy a → b)
⊕
(wait for a → c or c → b
 if a → c is available
   buy a → c; wait for c → b; buy c → b
 else if c → b is available
   buy c → b; wait for a → c; buy a → c)
```

Listing 1. Speculation. $\oplus$ denotes an exclusive choice construct.

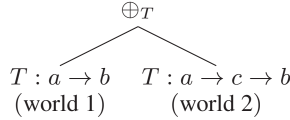


Figure 2. Tree Structure of T's Choices. $\oplus_T$ denotes the choice of T.

optimal. Rather than risking an incorrect choice, in this paper, we propose a new programming model intended to avoid sub-optimal choices where possible. In the earlier example, rather than having fixed resources, we can virtualize them. Suppose there are two *virtual* worlds, with *duplicated* resources. This allows the traveler to attempt both of the routes separately and simultaneously. One of the virtual worlds which succeeds can then be the state of the actual resources (we discuss this further in the following sections). We call this model of computation, *speculative nondeterminism*. It is nondeterministic because the business strategy of S varies, and the traveler T does not know which route will succeed, but has to execute both to find out. It deals with concurrency since there are many users interacting with S.

We now informally explain the idea of *speculation*. Assume that the traveler T uses speculation, i.e. tries routes $a \to b$ and $a \to c \to b$ separately (see Listing 1). Because of the two choices by T, we create two mutually isolated environments which we call *world 1* and *world 2* (see Figure 2). Each choice is mutually exclusive, for example, in world 1, he tries route $a \to b$; whereas in world 2, he tries route $a \to c \to b$. He has \$100 in each world and waits for the corresponding ticket(s). The actions performed by S are identical, so the availability of the tickets are the same. Therefore, as long as ticket $a \to b$ (or $a \to c$ and $c \to b$) eventually comes, T can buy the ticket(s) and succeed in world 1 (or world 2).

1.2 Exit

Suppose there is another traveler P who only wants to buy the ticket $a \to b$. Assume that T has already create two worlds, and since P has to interact with T, he has to duplicate itself and execute in both worlds. Even if P gets ticket $a \to b$ before T in both worlds, which virtual world is the *actual world* is still unknown as long as T is still executing his two choices. In this case, can P get out of the speculative computation with a *real* ticket?

We provide an *exit* mechanism for the speculative nondeterminism framework which allows an agent to return to the external computation outside the virtual worlds. Since speculation provides virtual computation in multiple worlds, we call the exit as *returning to (external) reality*. The idea is that a computation agent inside a world can only bring out certain information which it is concerned with and that information has to be consistent across all worlds. For example, P is only concerned about the ticket $a \to b$, so after P possesses the ticket in both worlds, P is allowed to leave by using exit, while the two worlds remain running for S and T.

1.3 Commit

Suppose there is another traveler Q who has identical choices as T, and Q is concerned with whether he has acquired enough tickets to go from a to b. When Q speculates in the same way as T, the combination of choices from both T and Q creates 4 virtual worlds in total (see Figure 3(a)). Assume that S sells only the ticket $a \to b$ now, and Q takes the ticket in both worlds w_1 and w_3 but Q's computation in worlds w_2 and w_4 is blocked waiting for ticket $c \to b$. Q cannot leave the speculation by using exit as it does not have the same ticket in all worlds.

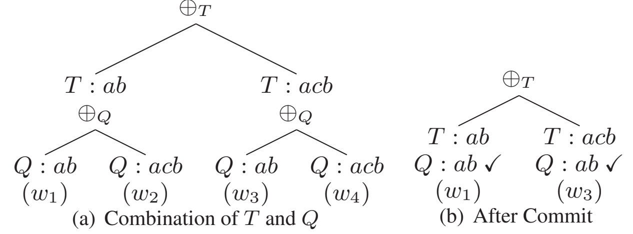


Figure 3. ab denotes the route $a \to b$ while acb denotes $a \to c \to b$. Check mark (✓) indicates the choice works.

If some worlds are pruned, that this can make it easier for an agent to exit as there are fewer worlds which need to be consistent. To prune the worlds, we introduce the *commit* operator ($\mathbf{cm}$) as shown in Listing 2.

By using $\mathbf{cm}$, it indicates that the current choice works satisfactorily and it's fine to prune the other choice. In the case of Q, once $\mathbf{cm}$ is reached in w_1 (or w_3), w_2 (or w_4) will be pruned. After both w_2 and w_4 are pruned, the tree of worlds is shown in Figure 3(b). Now what Q concerns is consistent in w_1 and w_3, so Q can exit from the virtual worlds.

$\mathbf{cm}$ not only helps the agent exit early, but also can be used together with sequential constructs to gain more programmability. For instance, consider Listing 3, when used together with sleep, the agent program can achieve some *preference* of cheaper multi-hop tickets ($a \to c \to b$ costs \$90) over more expensive single ticket ($a \to b$ costs \$100), because after the traveler possesses $a \to b$ in one world, during the time he is sleeping, he may be able to acquire both $a \to c$ and $c \to b$ in the other world, and commit in that world to prune the sleeping one.

Multiple worlds was proposed in [17] using a generalized committed choice (GCC) construct. In this paper, we generalize and extend GCC and propose a programmable concurrency control framework called *speculative nondeterminism*, which allows concurrent agents to

- *speculate* by specifying exclusive choices and executing these choices in combinatorial and mutually exclusive runtime environments called worlds;

```
(wait for a → b
 buy a → b
 cm)
⊕
(wait for a → c or c → b
 if a → c is available
   ...
 else if c → b is available
   ...
 cm)
```

Listing 2. Commit. $\mathbf{cm}$ is the commit operator. Omitted lines are the same as in Listing 1.

```
(wait for a → b
 buy a → b
 sleep for 10 seconds
 cm)
⊕
(wait for a → c or c → b
 ...
 cm)
```

Listing 3. Preference. Omitted lines are the same as in Listing 2.

- *exit* from the virtual worlds and return to reality even when other agents are still speculating; and
- control usage of system resources resulting from the choices using *commit* operators anywhere in the program.

Our framework provides a more practical realization of speculation than GCC. This paper has two main contributions:

1. We propose and *formalize* a concurrency framework that exploits combinatorial choices to improve the chances of success by agents in open and real-time applications.

   The formal definition allows us to *precisely* describe the operational semantics and reason about the properties and design choices in this framework. Moreover, the operational semantics is designed with practical implementability in mind.
2. We have *implemented* a proof-of-concept prototype system based on the framework showing that the system can effectively prune the computation space while achieving solutions in reasonable time.

The rest of this paper is structured as follows. Section 2 proposes a language for speculation with a concrete operational semantics. Section 3 describes a runtime system we built for the framework, and Section 4 evaluates the system using three benchmarks. Section 5 discusses related work and Section 6 concludes the paper.

2. Language for Speculation

In this section, we define a language for speculation – its syntax and operational semantics. We also present illustrative examples.

2.1 Data Model

In speculative nondeterminism, we assume a centralized data store acting as the primary programming environment. The implementation of this store is not part of the language and therefore independent from the operational semantics of the language. Here we present the abstract data model of the store that can be instantiated into different concrete models.

A data model is a 6-tuple

$$\mathcal{DM} = \langle D_0, \mathcal{T}, \mathcal{P}, \mathcal{D}, \vdash, \psi \rangle$$

where

D_0 defines an empty data store, for example, an empty set $\emptyset = \{\}$ or an empty multi-set $\{\!\!\{\}\!\!\}$.

$\mathcal{T}$ is the set of all possible data items in the data model. For example, $\mathcal{T} = \mathbb{N}$ for data models such as a set of natural numbers.

$\mathcal{P}$ is the set of all data operations available in the data model.

$\mathcal{D}$ is the set of all possible data stores. For example, $\mathcal{D} = 2^{\mathbb{N}}$ for data models such as a set of natural numbers.

$\vdash \subseteq \mathcal{D} \times \mathcal{P}$ is a binary relation denoted by $D \vdash d$. It defines when operation d can be executed on the data store D. For example, if the operation d has a precondition, $D \vdash d$ is true if and only if the condition is true when evaluated on the data store D.

$\psi : \mathcal{P} \times \mathcal{D} \to \mathcal{T} \times \mathcal{D}$ is the transition function that defines the effect of data operations on the store. In general, a data operation can do read, update, or both. For read purpose, a data item $t \in \mathcal{T}$ in the data store must be returned from ψ.

$$\begin{array}{rcl}
\text{Agent } a & := & p : f \mid \textbf{exiting} \mid \textbf{exited} \\
\text{Program } p & := & e \mid op.e \mid t.e \mid e_1.e_2 \mid \epsilon \\
\text{Operation } op & := & d \mid e_1 \oplus e_2 \mid \textbf{cm} \mid \textbf{cu} \mid \textbf{exit} \\
\\
\text{Tree } T & := & w \mid T_1 \oplus_k T_2 \\
\text{World } w & := & \langle A, D, S \rangle \\
\text{Agents } A & := & [] \mid [a_1, \ldots, a_m] \\
\text{Snapshots } S & := & \emptyset \mid \{s_1, \ldots, s_n\} \\
\text{Snapshot } s & := & \langle k, v \rangle
\end{array}$$

Figure 4. Syntax and Runtime Data Structures. $f : \mathcal{D} \to \mathbb{N}$ is a exit function. e denotes a local computation, $t \in \mathcal{T}$ is the input for the local computation, and ϵ is the end of a local computation. Dot (.) in p means sequencing. $d \in \mathcal{P}$ is a data operation while $D \in \mathcal{D}$ is a data store. $k \in \mathbb{N}$ is the identifier of an agent. $v \in \mathbb{N}$ is an exit value of an agent.

Next we give some examples of concrete data models under the above framework.

Tuple Space

The idea of tuple space comes from Linda [10], where a tuple space is a multi-set of tuples that can be accessed concurrently. A tuple is an ordered collection of fields. Agents can post their data to the tuple space in the form of tuples, and retrieve tuples as data from the tuple space that match a certain pattern. There are three major operations in the tuple space data model: (i) **in** reads and removes a tuple from a tuple space; (ii) **out** produces a tuple, writing it into a tuple space; (iii) **rd** non-destructively reads a tuple space and gets a copy of a tuple. Both **in** and **rd** are blocking while **out** is non-blocking. Formal definitions and operational semantics for the **in**/**out**/**rd** operations [5] can be easily adapted to this framework.

Key-value Store

Key-value store is a mapping from keys to values, e.g. $[a \mapsto 3, b \mapsto 5]$ is a key-value store where the key a gets value 3 and b gets 5. Typical data operations include: (i) creating a new key-value pair, (ii) updating an existing key with a new value, (iii) getting a value according to a key, and (iv) removing a key and its corresponding value. Conditional guards can also be combined with these data operations, e.g., $a > 4 \Rightarrow b \leftarrow 2$ waits (blocks) until the condition $a > 4$ is true in the store, and then updates the value of b to 2.

Other Models

Other more structured data models include relational data and logic programs. Data operations are insertion and deletion of tuples/predicates. Applications using these models are also typical.

2.2 Syntax

The syntax of the language for speculation is formally described in Figure 4. The idea is that we have two languages, one which deals with the speculative computation, the other is a host language where the speculation constructs are embedded within. The host and speculation language are conceptually orthogonal though in

```
in(ticket, ab)
⊕
(in(ticket, λx.x=ac ∨ x=cb) = (ticket, X)
 if X = ac
   in(ticket, cb)
 else
   in(ticket, ac))
```

Listing 4. Speculation Example using Tuple Space. We use `(...)` to denote tuples and $\lambda x.\varphi(x)$ denotes a condition for **in** and **rd** on the specific field in a tuple, where φ is the condition expression such as "x=`ac` $\vee$ x=`cb`" which means x is equal to either `ac` or `cb`.

practice one would want them to be closer. In our syntax, an agent consists of a program p and an exit function f. In the syntax in Figure 4, e is used to denote a language computation in the host language. Since the host language is arbitrary, we only define a meta syntax here[1] in that e is meant to evolve to some other local computation e' outside our semantics. We use ϵ to denote the end of such a local computation. There is a special syntax $t.e$ where t is intended to be the result of the data operation d (t is the result of transition function ψ). The meaning is that e is intended to consume the result t, which overloads, the dot symbol.[2] During runtime, an agent can also switch to special states **exiting** and **exit**, which will be explained in Section 2.4.

The top-level runtime data structure is the tree of worlds, while a world is a triple of a list of agents A, a data store D, and a set of snapshots S. A snapshot $s = \langle k, v\rangle$ is used in exit where k denotes the k-th agent and v is the exit value.

We use the tuple space data model throughout this paper due to its simplicity and expressive power. In order to explain further, we rewrite Listing 1 concretely using the tuple space data model and show it in Listing 4. We use this style of host language from now on to demonstrate local computations as well as to embed speculation primitives, where (i) identifiers starting with lowercase letters are symbol literals, (ii) identifiers starting with uppercase letters are variables, (iii) equal sign (=) denotes pattern matching and binding, (iv) variables are immutable after binding, (v) $\lambda x.\varphi(x)$ is a first-class anonymous function taking x as the parameter and evaluating $\varphi(x)$ as the result of the function on calling. Such features are derived from functional languages such as Haskell and Erlang. The choice of host language is primarily for ease of presentation, since the speculative nondeterminism framework is independent of the host language and local computations. However, we also have a prototype implementation in Erlang which also uses Erlang as the host language, thus our syntax is most similar to Erlang.

2.3 Examples

Next we describe some examples to help understanding the language.

Flight Reservation (FR)

In this example, we reuse the setting described in Section 1. There are two types of agents: ticket sellers and ticket buyers. Ticket sellers are airlines which put up the air tickets for sale. Ticket buyers are users who wants to travel from one city to another either on a direct flight or by several hops. There are five cities (a through e) and Figure 5 shows the flight connectivity map where an edge indicates that there is direct flight (in both directions) connecting two cities and the number on the edge is the price of the flight. In the seller program (Listing 5), the ticket seller loops for several times, each time picks an edge randomly, and sells a ticket corresponding to that edge (direction is also randomly selected). In the buyer program (Listing 6), every traveler (buyer) has exactly \$100 and tries to acquire enough tickets that would fly him or her from a to e so long as the total price is less than \$100. In other words, $a \to b \to e$ and $a \to c \to d \to e$ are both possible routes. `fly()` is defined using pattern matching on parameters. The base case is to arrive at the destination in the first fly, i.e. `fly(e, e, ...)` will evaluate to a symbol `arrived` since the starting point is the destination. The recursive case is `fly(a, e, ...)` will obtain unvisited neighbors and call `try()`. Note that using recursion with speculative nondeterminism is straightforward as it is considered as a local computation and local computations are neither limited nor affected by our framework.

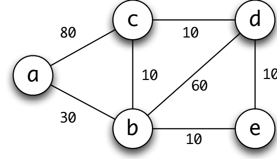


Figure 5. Flight Map and Cost Between 5 Cities

```
Ts = [[a,b,30],[a,c,80],[b,c,10],
      [b,d,60],[b,e,10],[c,d,10],[d,e,10]]
loop
  T = Ts[rand(6)]
  if rand(1) = 0
    out(ticket, T[0].T[1], T[2])
  else
    out(ticket, T[1].T[0], T[2])
exit
```

Listing 5. Ticket Seller's Agent. `[...]` denotes a list while `X[`i`]` is the i-th element of list `X`. `rand(`m`)` returns a random integer between 0 and m (inclusive). `X.Y` denotes the concatenation of symbols so that `u.v` evaluates to `uv`.

```
fly(X, X, Cash, V): arrived
fly(X, Y, Cash, V): Ns = unvisited_neighbors(X, V)
                    try(X, Ns, Y, Cash, V)

try(X, [N],    Y, Cash, V): buy(X, N, Y, Cash, V)
try(X, [N:Ns], Y, Cash, V): (buy(X, N, Y, Cash, V)
                            ⊕try(X, Ns, Y, Cash, V))
                            cm

buy(X, N, Y, Cash, V):
  in(ticket, X.N, λx.x ≤ Cash) = (_, _, Cost)
  fly(N, Y, Cash - Cost, [N:V])

fly(a, e, 100, [a])
exit
```

Listing 6. Ticket Buyer's Agent. Both `fly()` and `try()` are defined using pattern matching on parameters. `[N:Ns]` denotes a list which has `N` as its head and `Ns` as its tail. Underscore (_) matches anything in the context of pattern matching.

Stock Trading (ST)

In this example, we set up a stock market which follows the prices in the public exchange but is not openly available to the public. Traders can join the market at any time but their trades will not affect the public market. Our market starts with N stocks, each has a fixed amount of shares dedicated for this market. In general,

[1] The syntax here is similar to [5] which gives a semantics of Linda embedded in a host language.

[2] This is because d is defined in the data model but divorced from the host language, thus, there would be something in the host language to deal with the result t.

```
Me     = unique_account_name
Stocks = [...]  // a list of stock names
Start  = 5000   // start with cash of $5000
Goal   = 5500   // the goal is to end up with $5500

trade(Cash, α):
  X = Stocks[rand(N-1)]

  rd(price, X, _) = (_, _, P1)
  Q1 = ⌊Cash / P1⌋
  out(buy, X, Q1, Me)
  in(ack, Me, _, _) = (_, _, P2, Q2)
  C1 = Cash - P2 * Q2

  rd(price, X, λx.x ≥ (1 + α) × P2 ∨ x ≤ (1 − α) × P2)
  out(sell, X, Q2, Me)
  in(ack, Me, _) = (_, _, P3)
  C2 = C1 + P3 * Q2

  if C2 < Goal
    trade(C2, α)
  else
    cm

trade(Start, 0.01) ⊕ trade(Start, 0.05)
exit
```

Listing 7. Trader's Agent. Numbers are irrelevant and just for illustration purposes.

```
Me = unique_stock_name

serve(P, Q):
  case in(_, Me, _, _)
    (update, _, NewPrc, _): in(price, Me, _)
                            out(price, Me, NewPrc)
                            serve(NewPrc, Q)

    (buy, _, Q1, X): Q2 = min(Q1, Q)
                     out(ack, X, P, Q2)
                     serve(P, Q - Q2)

    (sell, _, Q2, X): out(ack, X, P)
                      serve(P, Q + Q2)

out(price, Me, Start_Price)
serve(Start_Price, Start_Quantity)
```

Listing 8. Stock-serving Agent. P is the current price of the stock while Q is the quantity of stocks available for sale.

each trader has a fixed amount of money in the beginning, and can buy (or sell) stocks from (or to) the market according to the public prices. Listing 7 shows the trader's agent program. Each trader trades several rounds until his cash value reaches the goal. For each round, the trader randomly picks a stock X, buys it as much as possible, and waits for the price of X to change. α is a value between 0 and 1 which indicates the profit/loss margin percentage. For example, if a trader buys X at a price of P, he sells it at a price of either higher than $(1+\alpha)\times$ P (to profit) or lower than $(1-\alpha)\times$ P (to prevent further loss). Different α values lead to different trading behaviors, and for this example, the traders speculate to be either conservative (smaller α) or aggressive (larger α). Note that there is no atomicity guarantee between checking the price of a stock and actually buying it, which is also the behavior of a real market. Also an intention to buy can be partially filled by the stock-serving agent (see Listing 8). The stock-serving agent also waits for price updating (i.e. `(update, ...)`) from the public exchange in a realtime fashion.

Dining Philosophers (DP)

Consider the well-known "Dining Philosophers Problem" [8] - there are three philosophers A, B and C, sitting around a dining table, with three forks x, y and z between them. This classic concurrency problem shows that deadlock happens with certain sequences of acquiring the forks, e.g., A acquires x, B acquires y and then C acquires z. The classic solution to this problem is using a synchronization mechanism such as a counting semaphore, imposing a partial order on acquiring the forks, or requiring one philosopher to be asymmetric. However, such kind of solution requires additional constraints to the problem. In this example, the speculation framework offers an alternative solution which is purely based on the definition of the problem. We extend the problem to N dining philosophers sitting around a table with N forks. For simplicity, each philosopher has a unique index ranging from 0 to $N-1$. Listing 9 shows the initialization of data stores for dining philosophers. Listing 10 shows the agent program for dining philosophers. `my_index()` returns the index of the current philosopher.

```
for I = 0 to N-1
  out(fork,I)
exit
```

Listing 9. Dining Philosophers (initialization)

```
L = my_index(); R = (L+1) % N
(in(fork,L); in(fork,R)) ⊕ (in(fork,R); in(fork,L))
cm
out(fork,L); out(fork,R)
exit
```

Listing 10. Dining Philosophers. **%** denotes the modulo operation. **;** separates multiple statements in one line. **out** is non-blocking in the tuple space data model, so we don't need choices for putting back forks.

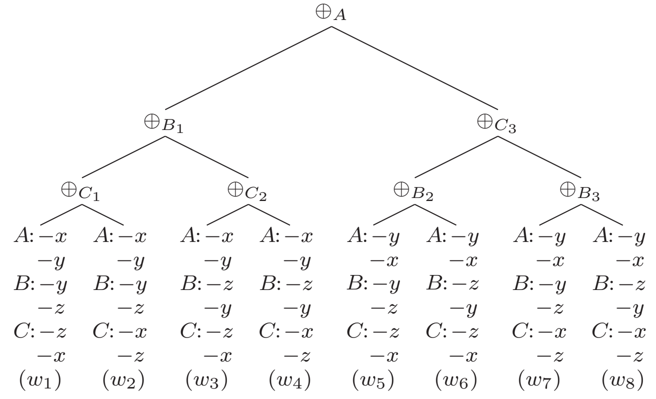


Figure 6. Three Dining Philosophers Problem (tree of choices). Minus sign ($-$) denotes taking the fork. Deadlock only happens in w_1 and w_8.

By using such kind of speculation, it can be guaranteed that there is always one world where nobody deadlocks. For example, assume $N=3$, the combination of the choices from three philosophers theoretically creates eight worlds as in Figure 6 (though in practice some of the worlds may never be there due to pruning).

There are many practical algorithms and applications (e.g., 2-phase locking) that are special cases or extensions of the Dining Philosopher Problem (DPP). So we generalize DPP as follows.

Definition 1 (Generalized Dining Philosopher Problem)**.** *Given a set of resources R which are available at the beginning, n agents each interested in a subset of resources $R_i \subseteq R (1 \le i \le n)$, and each agent is a two-phase process:*

1. *In the* acquisition phase *the agent consumes every resource $r \in R_i$ in any order;*

2. *In the* release phase *the agent puts back every resource $r \in R_i$ in any order.*

R_i's may overlap so the scheduler may produce a schedule which can cause deadlock among the agents.

Theorem 1. *Generalized Dining Philosopher Problems do not deadlock using speculative nondeterminism.*

Proof sketch. First, we construct a combination of choices which will never deadlock. Fix an ordering of the elements in R, for example $r_1, r_2, \ldots, r_m$ where $m = |R|$. For any agent i, it consumes R_i in this fixed order. Then there will be races instead of deadlocks because there will not be the case that agent i consumes r_x and waits for r_y, and agent j consumes r_y and waits for r_x ($x < y$ and $y < x$ cannot be true at the same time). Then it can be shown this combination will not be pruned by commits in an eventually blocking world due to the commit semantics described in rule CM. □

2.4 Semantics

The semantics of the speculation language is given in Figure 7.

Entrance New agents can dynamically enter the system at any time. ENTRANCE1 and ENTRANCE2 the addition of a new agent a to the system (denoted $\|$). The Plus sign $(+)$ denotes list concatenation. The whole tree is recursively traversed and a is appended to the agent list in every world.

Choice Symmetry SWAP shows that the two branches of an interior node in the tree are treated equally.

Data Operation An agent wants to execute operation d on the data store D (see DATAOP). This transition first has to meet condition $D \vdash d$. The effect of d on data store D is defined by the transition function ψ in the data model and it only affects the current world having store D without affecting other worlds in the tree. The result of d is denoted is t and provided to the remaining local computation e as a parameter (denoted by the special syntax $t.e$). $t.e$ is not processed by the speculation framework, but activates the local computation e and will eventually transform to another local computation e' (i.e. $t.e \longrightarrow_p e'$).

Local Computation The local computation transition is simply a placeholder for the semantics of the actual computation in the host language. As that is orthogonal to our semantics, the details are left unspecified. There is one special operation, $t.e$, which applies the result of a data operation t to some local computation e. For example, it might be to store t in a variable in the host language.

Choice and Commit Speculative nondeterminism provides for the creation and subsequent pruning of choices. CHOICE fires if an agent is reduced to a choice $(e_1 \oplus e_2).e$, and it consumes the choice construct in the agent to split the current world into two (i.e. T_1 and T_2). Section 2.5 discusses further the ramifications of commit, here, we start with first describing what the commit rules do. When the k-th agent executes **cm** in a leaf world of the tree structure, as shown in rule CM, the other side of the choice (i.e. T) is pruned only if the current choice node is $\oplus_k$. Similarly, **cu** is used where the agent wants to explicitly gives up the current choice branch, so the current side of the choice is pruned, as shown in rule CU. **cm** and **cu** are symmetric and both of them are *commit* operators. Commit of the k-th agent can only be executed in a world with $\oplus_k$ as its parent, and it is blocking when the parent node is not $\oplus_k$.

Exit In general, a single agent will have many versions of itself executing in all the multiple virtual worlds. Thus, if it were simply to exit from all the worlds on the first **exit** operation, this could lead to some form of data inconsistency. We deal with this using the exit function f specified by each agent which allows an agent-specific consistency condition to be used. Note that different agents may be interested in different aspects of the data store. When an agent exits, all the worlds should look the same from the perspective of that agent. However, another agent may not consider the

Entrance transition: $T \| a \Longrightarrow T'$

$$\langle A, D, S\rangle \| a \Longrightarrow \langle A + [a], D, S\rangle \qquad \text{(ENTRANCE1)}$$

$$T_1 \oplus_k T_2 \| a \Longrightarrow (T_1 \| a) \oplus_k (T_2 \| a) \qquad \text{(ENTRANCE2)}$$

Tree transition: $T \longrightarrow T'$

$$T_1 \oplus_k T_2 \longrightarrow T_2 \oplus_k T_1 \qquad \text{(SWAP)}$$

$$\frac{A[k] = d.e : f \qquad D \vdash d \qquad \langle t, D'\rangle = \psi(d, D)}{\langle A, D, S\rangle \longrightarrow \langle A[k \mapsto t.e : f], D', S\rangle} \qquad \text{(DATAOP)}$$

$$\frac{\begin{array}{c} A[k] = (e_1 \oplus e_2).e : f \\ T_1 = \langle A[k \mapsto e_1.e : f], D, S\rangle \\ T_2 = \langle A[k \mapsto e_2.e : f], D, S\rangle \end{array}}{\langle A, D, S\rangle \longrightarrow T_1 \oplus_k T_2} \qquad \text{(CHOICE)}$$

$$\frac{A[k] = \mathbf{cm}.e : f}{\langle A, D, S\rangle \oplus_k T \longrightarrow \langle A[k \mapsto e : f], D, S\rangle} \qquad \text{(CM)}$$

$$\frac{A[k] = \mathbf{cu}.e : f}{\langle A, D, S\rangle \oplus_k T \longrightarrow T} \qquad \text{(CU)}$$

$$\frac{A[k] = \mathbf{exit}.e : f \qquad s = \langle k, f(D)\rangle}{\langle A, D, S\rangle \longrightarrow \langle A[k \mapsto \mathbf{exiting}], D, S \cup \{s\}\rangle} \qquad \text{(EXIT1)}$$

$$\frac{\forall w \in leaves(T) : exiting(w, k) \wedge v = exitv(w, k)}{T \longrightarrow exit(T, k)} \qquad \text{(EXIT2)}$$

$$\frac{\begin{array}{c} w \in leaves(T) \\ \forall k : exiting(w, k) \vee exited(w, k) \end{array}}{T \longrightarrow w} \qquad \text{(COLLAPSE)}$$

Local computation transition: $e \longrightarrow_p e'$

$$\begin{array}{llll} e & \longrightarrow_p op.e' & e & \longrightarrow_p \epsilon \\ t.e & \longrightarrow_p e' & e_1.e_2 & \longrightarrow_p e_1'.e_2 \end{array}$$

Helper functions

$$\frac{T = T_1 \oplus_k T_2}{leaves(T) = leaves(T_1) \uplus leaves(T_2)}$$

$$leaves(w) = \{\!|w|\!\}$$

$$\frac{w = \langle A, D, S\rangle \qquad \langle k, v\rangle \in S}{exitv(w, k) = v}$$

$$\frac{w = \langle A, D, S\rangle}{exiting(w, k) = (A[k] = \mathbf{exiting})}$$

$$\frac{w = \langle A, D, S\rangle}{exited(w, k) = (A[k] = \mathbf{exited})}$$

$$\frac{T = T_1 \oplus_i T_2}{exit(T, k) = exit(T_1, k) \oplus_i exit(T_2, k)}$$

$$\frac{w = \langle A, D, S\rangle}{exit(w, k) = \langle A[k \mapsto \mathbf{exited}], D, S\rangle}$$

Figure 7. Semantics of the Speculation Language. $[\ldots]$ denotes a list, $A[k]$ is the k-th element in list A, and $A[k \mapsto x]$ denotes the updating of the k-th element to x.

worlds to be the same since it can have a different exit function. This also means that an agent properly exiting is only achieved when the consistency requirement is obtained.

Rule EXIT1 and EXIT2 shows the exit semantics. EXIT1 shows the case when an *instance* of the k-th agent reaches $\mathbf{exit}$ in a world. In this case, the exit function f is applied to the current data store D and a *snapshot* is created and added to S.

The exit function f takes a data store as input, and returns an integer as the evaluated result. The data store is just the one associated with the current world. According to the data store, the exit function is expected to produce an integer which can be a 0/1 indicator, a heuristic value, or an encoding of more complicated mathematical objects. For example, in the simplest case, $f(d) \equiv 1$ allows the agent to exit without any condition as long as it completes execution of all the operations in the program. The ability to return an integer provides more flexibility and enables complex exiting logic to be embedded in this exit function.

While taking a snapshot, the agent in that world switches to a special state $\mathbf{exiting}$. EXIT2 checks all the worlds in the tree. Only if (i) the k-th agent has switched to $\mathbf{exiting}$ state in all the worlds, and (ii) the exit value v of the k-th agent is *identical* across all the worlds, then the k-th agent can exit from the worlds at the same time. For EXIT2, the exit value v, which can be thought of as the return value from the speculative computation, will be eventually provided to a local computation $\tilde{e}$ which is completely outside the system of virtual worlds, and has no more speculation and data store operations. The idea of this type of consistent exit across all worlds is analogous algebraic factorization, i.e. $ab+ac = a(b+c)$.

Collapse Note that commit is optional in an agent program, so the agents can let the tree expand without pruning, and finally leave the tree without reducing to one world. Therefore some agents may never exit due to the inconsistency of some worlds, even if all the agents have finished execution. COLLAPSE handles this case to let these agents exit as well as to reclaim system resources. The system could employ a global collapse rule to pick any world where all agents have completed execution and reduce the tree to one world. For COLLAPSE, the exit value for every agent k is also returned in this way. $\tilde{e}$ can make use of the exit value to extract what the agent concerns.

Helper Functions We also define a set of helper functions to simplify the semantics.

$leaves(T)$ returns a multi-set of the leaf worlds in tree T.

$exitv(w, k)$ retrieves the exit value, i.e. the evaluation result of the exit function, of the k-th agent in world w.

$exiting(w, k)$ is a predicate indicating whether the k-th agent in world w is in the special state of $\mathbf{exiting}$.

$exited(w, k)$ is a predicate indicating whether the k-th agent in world w is in the special state of $\mathbf{exited}$.

$exit(T, k)$ switches the k-th agent in all the worlds in T to the special state of $\mathbf{exited}$.

2.5 More on Commit

Commit is a special and important operation in speculative nondeterminism because: (i) it gives the agent the power to specify preferences among different choices, e.g., in Listing 3; (ii) it prunes some of the virtual worlds to make it easier for agents to exit because the fewer remaining worlds means easier consistency checks, and thus it improves the responsiveness and overall throughput of the system; (iii) by pruning worlds, it also reclaims precious system resources such as memory and CPU cycles, which is critical to the viability of the multi-world speculation , even though the naive combinatorial problem space is exponential and prohibitive.

However, it is also important to understand that whenever commit is used and pruning is done, potential solutions can be pruned away, and deadlock or blocking may arise as a result. So from the system point of view, we design a commit semantics that is *localized* and less aggressive because the scope of the pruning is restricted to be of height 1 only and affects only a small fraction of the whole tree (see Rule CM). This commit semantics is also more efficient because pruning decisions are made locally. There are of course other possible commit semantics [17] which are more eager in pruning. For example, one type of commit requires agent T to commit in all left or right subtree rooted at $\oplus_T$. These are either too aggressive and lose too many solutions or require coordination among various worlds which is more costly to execute in practice.

From the programer's point of view, she should use commit with care, knowing that without commit, the multi-world space grows very quickly and the program probably cannot scale, but with commit, the program could potentially lose valuable solution. It makes more sense to put commit later rather than early in the choice. If one must put a commit in the middle of a sequence of operations like $op_1.op_2.\mathbf{cm}.op_3.op_4$, she should ensure that the remaining operations after $\mathbf{cm}$, i.e., op_3 and op_4, are not likely to block, e.g., when they are local computations.

3. System Implementation

We have implemented a proof-of-concept system using Erlang, to show the viability of the speculative nondeterminism in a practical setting. Both the system and the agent programs are written in Erlang, and communicate by using global registered names and message passing. Every node in the tree of worlds is a process [3] in Erlang VM. Different instances of the same agent are also different processes. Agent instances talk to the worlds they live in directly by message passing. The system also assumes a tuple space data model and uses data operations similar to Linda, such as $\mathbf{in}/\mathbf{out}/\mathbf{rd}$. Tuples in the a tuple space are exactly Erlang tuples, while the tuple space is a list of tuples along with additional indexing structures. $\mathbf{in}/\mathbf{rd}$ conditions are represented as Erlang anonymous functions (i.e. `fun(...) -> ... end`). We also defined a set of macros in Erlang to help building conditions. For example, `?Any` is defined as `fun(_) -> true end`, `?LessThan(X)` is defined as `fun(V) -> V < X end`, etc.

Our system benefits from Erlang in two aspects: (i) Processes in Erlang are not native OS processes/threads, but *ultra-lightweight processes* with extremely fast creation and tiny memory footprint. There are large amount of world splitting and pruning operations in runtime. By reducing the overhead caused by the system, agent programs can have more CPU time and thus more opportunities to capture their needs in realtime. Tiny memory footprint allows us to run millions of processes at the same time even on a single machine, which provides the possibility of running large scale applications. (ii) Erlang processes share no memory, and the communication model among them is asynchronous *message passing*. Erlang guarantees causality and eventually delivery, provided that the recipient exists, and provides process linking and monitoring functionality such as `erlang:link/1` and `erlang:monitor/2`. These guarantees and facilities help the system being robust and well-structured and thus provide possibilities of customizability and extensibility.

Next we describe a few optimizations that make the implementation more practical.

[3] By using the word *process* in this section, we mean the ultra-lightweight processes in the Erlang virtual machine, while *native processes/threads* mean OS processes/threads

Lazy Forking

Lazy forking is a *fork-on-need* mechanism which enables *asynchronous coordination* of the agents. For example, initially program P lives in world w. Then program Q joins and creates two choices in world w, thus conceptually splitting w into two descendant worlds w_1 and w_2, and Q itself forks into two processes. In a naïve implementation, P has to fork into two processes as well, each corresponds to a program instance in one of the new worlds. By lazy forking, P remains one process at this point, and delays the forking until it starts interacting with the store. This mechanism helps control the excessive forking and the number of running processes in the system.

Exponential Backoff of Splitting

In some cases, a large number of agents issue the choice construct in the same world simultaneously. Naïvely splitting this world in this case doesn't do any real work but can easily get stuck in the exponentiality. From the perspective of an agent instance a, the world w it lives in is stored locally and should be a leaf in the tree of worlds. However, because of the lazy forking mechanism described above, that world w may be already split into two descendant worlds w_1 and w_2,. When a wants to issue the choice construct in this case, the system uses exponential backoff to delay the choice creation, and later allows a to split in the two descendant worlds of w. The delay gets exponential when w_1 (or w_2) is also split at the time a is allowed to split it. There is also a limit of the delay so agents can have reasonable responsiveness. This mechanism injects nondeterminism into the scheduler so that programs that are just about to commit can commit first and start pruning the worlds.

Condition Triggering

Both **in** and **rd** in the tuple space data model are blocking on the condition. When there's an update to a world's tuple space, instead of waking up all the agents in the world to check the condition, the system only wakes up part of them by using triggering. In our prototype system, the triggering is enabled by *indexing* the fields in the tuples, and is available for tuple operations **in** and **rd**.

Tree Partitioning

Besides the optimizations for reducing overhead, we also provide a *tree partitioning* mechanism in order to reduce exponentiality. When there are large number of agents, not all of them are interested in the same kind of resources. For n agents with no interleaving interests and each with 2 choices, naïvely there should be 2^n worlds. However, since their interests are non-interleaving, we only need $2n$ worlds corresponding to their $2n$ choices in total. The tree partitioning mechanism works in such a way that each agent starts with its own world, instead of joining the leaf worlds of a common tree. As long as the agent doesn't have interest interleaving with other agents at this point of time, it splits and prunes on its own tree. In this way, agents with non-interleaving interests are separated and partitioned into different trees. When at some point of time, an agent a_i starts to be interested in some resources which some other agent a_j is also interested in, the system merges the trees of a_i and a_j together. Suppose the trees of a_i and a_j have p and q leaf worlds respectively, the resulting tree after merging them together will have exactly pq leaf worlds to represent the combination of choices.

In order to concretely illustrate the use of tree partitioning, we reuse the FR example described in Section 2.3. Suppose, in Figure 5, a traveler T_1 wants to buy either $a \rightarrow b$ or $a \rightarrow c \rightarrow b$, while another traveler T_2 wants to buy either $d \rightarrow e$ or $d \rightarrow b \rightarrow e$. Then $R_1 = \{ab, ac, cb\}$ are resources that T_1 is interested in, and $R_2\{de, db, be\}$ are resources that T_2 is interested in. For T_1 and T_2 there is no need to "multiply" their choices together. Also suppose

```
Ts = ...  // tickets as in Listing 5
loop
  T = Ts[rand(6)]
  if rand(1) = 0
    spawn_process{out(ticket, T[0].T[1], T[2])
                  exit}
  else
    spawn_process{out(ticket, T[1].T[0], T[2])
                  exit}
```

Listing 11. Ticket Seller's Program Making Use of Tree Partitioning. `spawn_process()` creates a short-lived actor for interacting with the system of virtual worlds. $\{\ldots\}$ denotes a separate program code instead of evaluation.

the agents with interleaving interests in R_1 and R_2 create a tree of p and q leaf worlds respectively. By using tree partitioning, we only get $p + q$ instead of pq worlds.

Intuitively, tickets between different cities should be separated in this way. However, in Listing 5, the ticket seller's agent touches every ticket, so it's considered by the system to be interested in both $ab \in R_1$ and $de \in R_2$. In this case, two trees containing T_1 and T_2 will be merged together. In order to benefit from tree partitioning, the ticket seller should be moved out of the system and creates a short-lived actor for each of the tickets (see Listing 11). There's no need to modify the ticket buyer's agent program.

4. Evaluation

In this section, we evaluate our proof-of-concept system for speculative nondeterminism by using benchmarks based on the three examples described in Section 2.3.

Flight Reservation (FR) We use the same strategy described in Listing 5 and 6, but on a larger airline map [4] of 30 international cities.

Stock Trading (ST) We use the same strategy described in Listing 7, but limit the number of trades to be exactly 3. The total number of stocks is 10 and the initial available shares are proportional to the number of traders. For each stock, we replay 1000 data points which are real stock market data in the U.S.

Dining Philosophers (DP) This example in the benchmark is exactly the same as shown in Listing 9 and 10.

We evaluate our runtime system running these benchmarks on Linux (2.6.18) with Intel Xeon E5520 2.2GHz CPU and 24GB RAM. Both the prototype system and benchmark programs are written in Erlang, and run in Erlang VM (64-bit R15B02) with HiPE and SMP (4 schedulers) enabled.

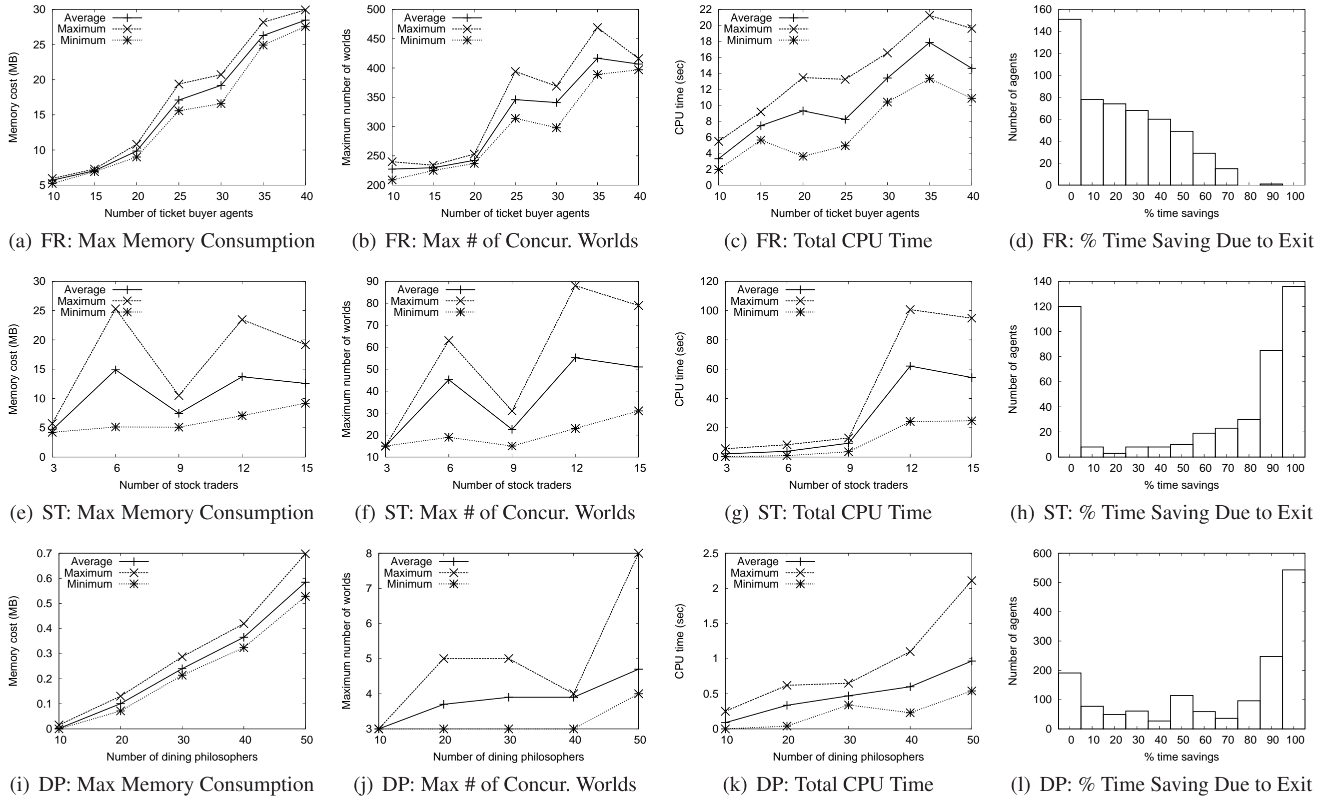
In each benchmark, we conducted several experiments with different number of concurrent agents. We ran each experiment 10 times and record the average, maximum and minimum values for three key metrics: maximum memory consumption, maximum number of simultaneous worlds and total CPU time. In addition, we record the percentage time savings of each agent by

$$\frac{\text{exit time of last agent in system} - \text{exit time of this agent}}{\text{exit time of last agent in system}}$$

because without exit mechanism in the system, all agents would have waited till the last agent completes to exit from the system together.

Figure 8 shows the results. We can see that even though theoretically the maximum number of worlds in these examples are

[4] We use the HA30 dataset which is available at `http://people.sc.fsu.edu/~jburkardt/datasets/cities/cities.html`.



(a) FR: Max Memory Consumption (b) FR: Max # of Concur. Worlds (c) FR: Total CPU Time (d) FR: % Time Saving Due to Exit

(e) ST: Max Memory Consumption (f) ST: Max # of Concur. Worlds (g) ST: Total CPU Time (h) ST: % Time Saving Due to Exit

(i) DP: Max Memory Consumption (j) DP: Max # of Concur. Worlds (k) DP: Total CPU Time (l) DP: % Time Saving Due to Exit

Figure 8. Evaluation Results

prohibitively large (i.e., 2^n), in reality, with the help of the commit operators, we can keep the size of the universe as well as other computation costs down to very manageable. In other words, given well programed agents and a reasonable commit mechanism, the prototype system can manage the exponentiality inherent in the problems while achieving solutions for all agents in reasonable time. However, as a proof-of-concept prototype, the system can also be affected by some special combination of choices and realtime events. For example, in Figure 8(e) and 8(f) of the ST benchmark, the anomalies can be caused by a special combination of trading strategies which leads to resource contention. Different number of agents also slightly affects the scheduling so some realtime price updates can be missed by the agents.

Furthermore, Figure 8(d), 8(h) and 8(l) show the distributions of agents on the time savings due to the exit semantics. Most of the agents save at least half of the overall time with the exit semantics implemented. FR has slightly less savings in time because most of the worlds only have limited resources for the agents to commit. Overall, the results clearly demonstrate that the exit mechanism significantly improves the user responsiveness and the efficiency of the computation. A carefully engineered application which uses the speculative nondeterminism model should be even more efficient.

5. Related Work

Generalized committed choice [16, 17] offers the preliminary and early ideas of speculation. This paper differs in a number of ways: (i) we propose a simpler and more practical language that is geared toward actual implementation; (ii) the new commit operator is more conservative and hence reduces the chance of removing potential solutions; (iii) we deal with agents exiting the system; (iv) we formalize the operational semantics in a complete calculus which allows us to reason about the system; and (iv) most importantly, we demonstrate the viability of speculative nondeterminism with a prototype implementation.

The most closely related work to this paper is Shared Prolog [4], which is in the family of concurrent logic programming languages [22]. Other related languages include Concurrent Prolog [23] and Guarded Horn Clauses [24]. These languages extend traditional logic programming languages with some form of guards [7] which serves as a synchronization construct for concurrency. The guards function intuitively as read-only goals to choose between alternative sequential logic programs. The key difference is while there may be nondeterministic choices (OR-parallelism) in a sequential program, these choices are not exposed to the other concurrently executing logic programs and hence there's no combinatorial choices which is the key artifacts of speculative nondeterminism. Shared Prolog uses the blackboard data model to implement the guards. These guards only read or remove tuples from the blackboard but not add tuples. This model is designed for collaborative applications in which agents share tasks, but is less conducive to a competitive environment where speculative nondeterminism is more suitable. Our implementation of the data model is inspired by Linda [12], a well-known blackboard model. Later work on multiple tuple spaces in Linda [11] is also related to the multi-world model in our framework. However, speculative nondeterminism can be applied to a general data store, and not just blackboards.

Deep guards in the constraint programming language Oz [18, 21] are also similar to the concurrent logic programming languages. It supports local computation spaces with monotonic constraint stores where all local effects only become globally visible at the

time of commit. Our framework, in contrast, allows arbitrary updates to the data store where agents concurrently interact with one another both inside and outside choices. Viewing the operations before the commit as a guard, we allow updates in the guard which is compatible with regular imperative programming languages while monotonic stores are not.

The idea of speculation has been used in programming languages in more restricted contexts. Language constructs such as `future` in Multilisp [19] and Java [26] and speculative composition/iterations in C# [20] are also examples of programmable speculative parallelism. As Osborne rightly pointed out [19], speculation requires: a means to control computation to favor most promising ones; and a means to abort computation and reclaim the resources. The commit and exit semantics in our framework does this in general way for arbitrary agent programs. Recent versions of Glasgow Haskell Compiler support speculative execution with an abortion mechanism to back out in case of a bad choice called optimistic execution [9]. In composable memory transactions (CMT) [14], the `orElse` construct specifies several alternatives in memory transactions. However, our framework is more general, dealing with *open* agents with arbitrary imperative programs, concurrency, reasoning with choices and also allows real-time programming. Speculation has been used for extracting instruction-level or thread-level parallelism [13]. Here, idle hardware units are used to execute code that might be useful later. In case the speculation is wrong, hardware support is need to cancel the side effects. One form of such speculation is known as *eager execution* [25] where both sides of a conditional (`if-then-else`) is executed before the condition predicate is evaluated. However, these forms of speculation are in restricted contexts and are usually intended for performance rather than expressivity.

Nondeterminism has long been studied in programming languages. While there are many forms of nondeterminism, we can characterize two classes, don't know nondeterminism and don't care nondeterminism. Guarded languages such as Dijkstra's guarded commands [7], CSP [15] and the committed choice languages mentioned earlier, fall into the don't care class. Logic Programming or Constraint Programming languages, on the other hand, are more concerned with don't know since the objective is to find the solution to a program which involves combinatorial reasoning. Speculative nondeterminism also deals with don't know nondeterminism but adds concurrency and interacting agents into the mix. Since we have concurrency, we can also express don't care nondeterminism. The cut (`!`) mechanism in Prolog [6] is related to our commit operator as both prune the solution space. Unlike cut, commit is more conservative and do not always prune.

Finally, the multiple-world model is also related to multiversion concurrency control (MVCC) [2, 3] in databases. MVCC keeps multiple versions of a database in the sense that each user has a unique view of the database, and updates are not visible to other users until the transaction has committed. Transactions that are deemed invalid are discarded which is similar to pruning worlds in our framework. Implementations of transactional memory also use this idea. While most of the work in this space are in concurrency control, this paper proposes a *programmable* framework for a new kind of nondeterminism.

6. Conclusion

Speculation and nondeterminism are not new to computing. In this work, we present a novel framework that joins the two concepts together in a programmable concurrency control model. This framework allows users to specify exclusive choices in their program and models the combinatorial choices by multiple programs in multiple virtual worlds. The advantage of this is that by attempt all this possibilities, users have much better chance of achieving their goals and the system can produce better throughput with less deadlocks and starvation. We have showed that the theoretical exponentiality in the combinatorial computation can be effectively controlled by a commit operator and the program termination and return can be accelerated using an exit mechanism.

References

[1] D. M. Armstrong. *A Combinatorial Theory of Possibility*. 1989.

[2] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[3] A. Bestavros and S. Braoudakis. Timeliness via speculation for real-time databases. In *IEEE Real-Time Systems Symposium*, pages 36–45, 1994.

[4] A. Brogi and P. Ciancarini. The concurrent language, shared prolog. *ACM Trans. Program. Lang. Syst.*, 13(1):99–123, 1991.

[5] P. Ciancarini, K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 77–106. 1995.

[6] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3), 2001.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[8] E. W. Dijkstra. Hierarchical ordering of sequential processes. In *The origin of concurrent programming*, pages 198–227. 2002.

[9] R. Ennals and S. L. P. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP*, 2003.

[10] D. Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, 1985.

[11] D. Gelernter. Multiple tuple spaces in linda. *PARLE'89 Parallel Architectures and Languages Europe*, pages 20–27, 1989.

[12] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992.

[13] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, 1998.

[14] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of PPoPP*, 2005.

[15] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[16] J. Jaffar, R. H. C. Yap, and K. Q. Zhu. Coordination of many agents. In *Proceedings of the 21st International Conference on Logic Programming, ICLP*, 2005.

[17] J. Jaffar, R. H. C. Yap, and K. Q. Zhu. Generalized committed choice. In *COORDINATION*, pages 191–210, 2007.

[18] M. Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards*. PhD thesis, U. of Saarlandes, 1999.

[19] R. B. Osborne. Speculative computation in multilisp. In *Proceedings of US/Japan Workshop on Parallel Lisp*, 1989.

[20] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI*, pages 50–61, 2010.

[21] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In *PPCP*, pages 134–150, 1994.

[22] E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[23] E. Y. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. *New Generation Comput.*, 1(1):25–48, 1983.

[24] K. Ueda. Guarded horn clauses. In *Logic Programming '85, Proceedings of the 4th Conference*, pages 168–179, 1986.

[25] A. K. Uht, V. Sindagi, and K. Hall. Disjoint eager execution: an optimal form of speculative execution. In *MICRO*, pages 313–325, 1995.

[26] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for java. In *OOPSLA*, pages 439–453, 2005.