A Runtime System for Generalized Committed Choice

Xiao Jia Kenny Q. Zhu
Shanghai Jiao Tong University
{xjia,kzhu}@cs.sjtu.edu.cn

Joxan Jaffar Roland H.C. Yap
National University of Singapore
{joxan,ryap}@comp.nus.edu.sg

Abstract

Traditional nondeterministic programming constructs (Dijkstra guards, CCP [6] and deep guards [10]) do not allow operations which modify the runtime environment without committing to a particular alternative. Generalized committed choice (GCC) allows speculative computations across different alternatives to execute in parallel and isolation. Speculation implicitly forks an environment into separate ones for the alternatives and later one of these environments can be committed to [3]. Speculations from concurrent processes can nevertheless interleave and synchronize against each other. In this paper, we present a concrete architecture, its implementation and optimizations for GCC where the store of the environment is in the form of record spaces. Our prototype implementation allows GCC to be embedded in traditional languages such as C/C++. Preliminary experimental results show that our runtime and GCC extension is a suitable coordination language for programming multiple distributed agents which employ speculation and choices.

1. Introduction

A coordination language embodies a coordination model and is responsible for the creation of, and the support of communication among, computational activities. The generalized committed choice (GCC) is a coordination model of a don't-know nondeterminism choice construct which allows the commit to occur anywhere within the choice [3] and strives for maximal inter-play of choices between agents to achieve beneficial speculation. GCC was proposed as a coordination model for speculation among concurrent agents. In this short paper, we report our preliminary progress in developing a concrete GCC runtime system. This work has the following contributions:

- We demonstrate GCC as a usable and practical coordination model for traditional programming languages such as C/C++ by embedding GCC constructs in regular C/C++ programs and by employing record stores as the data model;
- We extend the multi-world concept [3] to the notion of *multiple universe* which is transparent to programmers and crucial to handling the exponential explosion of multiple worlds;
- We show the viability of our system by two non-trivial examples and their runtime statistics.

The remainder of this paper is structured as follows. The rest of this section gives an overview of the GCC model and introduces the usage in concrete actual agent programs. Section 2 presents the overall system architecture. Section 3 discusses several optimization considerations. Section 4 gives some preliminary experimental results which show that speculative computation with GCC agents is practical.

1.1 Motivation

We use the motivating example for GCC [3]. Bob and Jill participate in an online trading system. Bob wants to upgrade his camera and Jill wants to downgrade. Their preferences which are exclusive (denoted by `XOR`) are:

Bob's agent: (sequencing is denoted by ';')

```
(buy(goodlens); sell(averagelens)) XOR
(buy(goodcam); sell(averagecam))
```

Jill's agent:

```
(sell(goodlens); buy(averagecam)) XOR
(sell(goodcam); buy(averagecam))
```

`Buy` and `sell` are synchronous actions which block until there is a matching transaction. Bob and Jill are not aware of each other, i.e. each agent acts independently. The objective is to satisfy as many agents as possible. In this example, GCC does this by creating four separate and isolated worlds arising from the potential interaction between the combination of Bob and Jill's choices as the agent computation proceeds. Figure 1 shows the effect of multiple worlds where `lens()` denotes buying(selling) lens by Bob(Jill), and `cam()` denotes buying(selling) camera by Bob(Jill). In this example, only in world w_4 can both agents be satisfied.[1] This

[1] w_1 cannot complete after the lens transaction while w_2 and w_3 block.

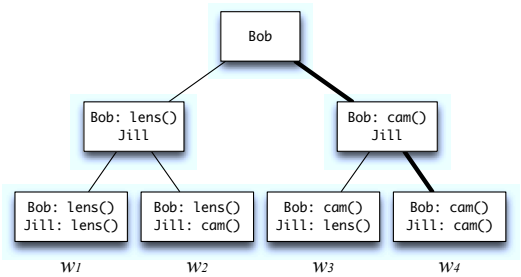


Figure 1: The multiple worlds in Bob-Jill example

requires speculation where agents can continue to interact. The runtime system in this paper provides essential facilities for embedding and implementing this example in conventional languages such as C/C++.

1.2 C Implementation of Bob-Jill Example

We show the above example in the APIs in our runtime system – the agents are `bob.cc` and `jill.cc`. Note that although the agents employ speculation and concurrency using GCC semantics, the embedding allows the agents to be written in vanilla-looking C (see Listing 1 and 2) using the following GCC coordination constructs in the runtime system:

`gcc`	fork multiple choices
`cm`	commit me (commit to this choice)
`cu`	commit you (kill this choice)
`in`	reads and removes a record from a record space
`rd`	non-destructively reads a record space
`out`	produces a record, writing it into a record space

Listing 1: `bob.cc`

```
1 void lens()
2   { buy(GoodLens); sell(AvgLens); cm(); }
3 void cam()
4   { buy(GoodCam); sell(AvgCam); cm(); }
5 int main()
6   { gcc(lens, cam); return 0; }
```

Listing 2: `jill.cc`

```
1 void lens()
2   { sell(GoodLens); buy(AvgCam); cm(); }
3 void cam()
4   { sell(GoodCam); buy(AvgCam); cm(); }
5 int main()
6   { gcc(lens, cam); return 0; }
```

Listing 3: Common routines for trading

```
1 enum Product
2     { GoodLens, AvgLens, GoodCam, AvgCam };
3 enum Type { Ack, Buy, Sell };
4 struct Offer { Type s; Product p; int i; };
5 void parse(struct Offer *, char *) { /*...*/ }
6 void sell(Product p)
7 { int id = generate_unique_id();
8   out("s=%d,p=%d,i=%d", Sell, p, id);
9   in("s=%d & i=%d & p?=1", Buy, id);
10  out("s=%d,p=%d,i=%d", Ack, p, id);    }
11 void buy(Product p)
12 { struct Offer f;
13   parse(&f,
14     in("s=%d & p=%d & i?=1", Sell, p));
15   out("s=%d,p=%d,i=%d", Buy, p, f.i);
16   rd("s=%d & i=%d & p?=1", Ack, f.i);   }
```

Common trading routines are in Listing 3. A trading transaction is a record of three elements: the trade type s, which can be `Ack`, `Buy`, or `Sell`; the product type p, which can be `GoodLens`, `AvgLens`, `GoodCam`, or `AvgCam`; and the transaction number i, which is generated and guaranteed to be unique. When an agent wants to sell some product p^*, he produces a record `(s=Sell, p=`p^*`, i=id)` (line 8), where id is a transaction number. Then he waits for someone to buy p^* by `in(s=Buy & i=id & p?=1)` (line 9). When an agent wants to buy some product p^*, he consumes a record by `in(s=Sell & p=`p^* `& i?=1)` first (line 14), and then `out(s=Buy, p=`p^*`, i=id)` (line 15). At the end of a transaction, the seller produces an `Ack` record (line 10) and the buyer reads it for acknowledgement (line 16).

1.3 Related Work

Earlier work [3] lays the theoretical foundation for GCC without addressing the practical details of employing GCC in conventional programming languages as well as a practical data model for real-world applications. Our runtime system uses a data model for GCC called *record space*, which uses Linda style tuples [2] with labels for tuple elements.

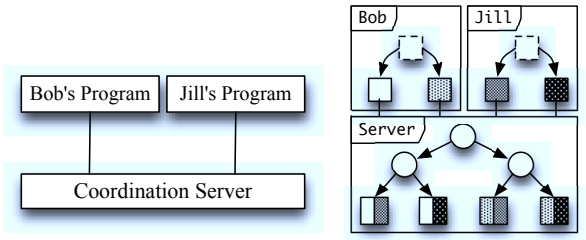
Logic programming languages, e.g. Prolog, search the solution space in a depth-first fashion. Shared Prolog [1] programs are composed of a set of parallel agents that are Prolog programs extended by a guard mechanism. Coordination of the agents is controlled by programmers via a centralized data structure roughly resembling a blackboard system [5].

Deep guards [7, 10] in Oz [8, 9] and its implementations, such as LVM [4], only enable local computation spaces with monotonic constraint stores and short-lived actors. All local effects only become globally visible at the time of commit. Such restrictions make it useless in the context of the applications considered here such as real-time marketplaces with long transactions.

2. System Architecture

Agents can be distributed and communicate through the *coordination server* as shown in Fig. 2a. (Our prototype uses agents in Unix). Two agents are initially two Unix processes which can fork into multiple processes if they involve choices.

Fig. 2b depicts the logical view of runtime environment. Bob's program starts from a single process and then forks into two processes for the two choices (to buy good lens or good camera). Jill's program works similarly. The coordination server maintains all runtime information, such as the



(a) Physical deployment view (b) Logical runtime view

Figure 2: Logical runtime view

structure of all the worlds and the relationships among them. Each process in the agent's environment conceptually lives in one or more worlds. Processes from different agents live in the same world if they interact. There is a communication channel between each process and the server.

3. Implementation and Optimizations

The main technical challenge in the implementation of GCC system is to contain the exponential growth of worlds because each world consumes valuable system resources. See [3] for more details on the conceptual semantics of GCC. In this section, we will present one key implementation idea that tackle the challenge and then briefly discuss several other optimizations.

3.1 Multiple Universe

A *universe* is a rooted tree of worlds along with their records. The records in a universe form a logical *partition* of the record space by record types. Agents that operate on different record types live in different universes. Worlds only get multiplied if two agents with choices are interested in the same type of records.

At startup, an agent program lives outside any universe. Once the agent wants to operate on some type of records, it either migrates to the universe which owns that record type, or creates a new universe with that type. Let U be a universe, $S(U)$ be the set of record types owned by the universe U. Suppose there are in total n universes, $U_1, U_2, \ldots, U_n$, in the runtime environment. When an agent program P in universe $U_i (1 \leq i \leq n)$ is about to operate on some record type T (by `in`/`rd`/`out`), there are three cases:

- $T \in S(U_i)$, then P operates on U_i as normal;
- $\exists 1 \leq j \leq n, j \neq i$, such that $T \in S(U_j)$, then U_i and U_j are removed and replaced by $U_k = J(U_i, U_j)$ where J joins two universes together to form another tree of worlds. U_k is then added to the runtime environment for P to operate on;
- $\forall 1 \leq j \leq n, T \notin S(U_j)$, then $S(U_i) \leftarrow S(U_i) \cup \{T\}$ and P operates on U_i.

The type of a record is the set of labels of the elements in the record. For example, the record `(a=1, b=2)` has the type $\{a, b\}$. Querying a record with constraints uses the label types, e.g. querying only on a, the condition becomes `(`p `& b?=1)` for `in`/`rd` operations, where p is a condition on a. The question mark ? is the *existence operator* which evaluates to 1 if the label exists, or 0 otherwise.

3.2 Other Optimizations

Lazy forking allows *asynchronous coordination* among agent programs and the server, and implies a *fork-on-need* mechanism. For example, program P starts with world w, which is later split into two worlds, x and y, by another program Q, which has to fork, but P doesn't need to, until it interacts with w.

Opportunistic forking is introduced that when there are too many `gcc()` requests at the same time, some of them will be delayed randomly in order to limit the number of worlds.

Condition triggering , which is enabled by *indexing* of record labels, is available for record consuming and reading actions (`in` and `rd`). Options for `in` and `rd` operations are: non-blocking, time-out and block until a condition.

Storage sharing works as a storage hierarchy where records are stored as closer to the root as possible. New records are always written to the current world. Searching a record starts from the current world, and if not found, the searching continues in the parent world, etc.

4. Evaluation

We evaluate our runtime system by running two examples which exemplify speculative computation inside choices. The experiments ran on Linux on a 4-core 2.0GHz Intel Xeon CPU with 8GB RAM.

Bob-Jill example : 10 instances each of Bob and Jill agents;

Flight reservation example : 10 instances of selling agents (Listing 4) and 20 instances of buying agents (Listing 5).

We propose the flight reservation example as a potential application of GCC. Here flight agents sell tickets whenever they are available, and travelers are ticket buyers waiting for desired tickets. Travelers may want to buy a multi-leg ticket at a lower price. For example, as shown in Fig. 3, a traveler from a to c may choose the path $a-b-c$ instead of the path $a-c$ because the former is cheaper. However, availability of the tickets is dynamic and not known in advance (real world ticketing is similar but airlines oversell). So if the traveler chooses a "bet-and-risk-it" strategy, he may not be able to get the two tickets ($a-b$ and $b-c$) in reasonable time. With the help of GCC, he can speculate in two worlds; in one world, he waits for the ticket $a-c$, and in the other world, he waits for the tickets $a-b$ and $b-c$. For this evaluation,

all the traveler agents start with 100 credits and try to buy tickets in order to reach d from a.

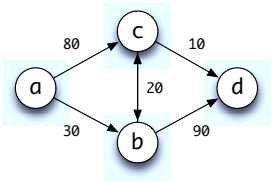


Figure 3: Costs between cities in the flight example

Listing 4: Ticket seller's agent

```
enum { N = 7 };
int t[N][3] =
{ { 2,3,10 },{ 0,1,30 },{ 1,2,20 },{ 2,1,20 },
  { 0,2,80 },{ 1,3,90 },{ 2,3,10 } };
int main() {
  for (int i = 0; i < N; i++)
    out("from_%d=1,to_%d=1,price=%d",
        t[i][0], t[i][1], t[i][2]);
  return 0;
}
```

Listing 5: Ticket buyer's agent

```
enum { N = 4 };
void wait_for_ticket(func_param p) {
  flt &d = *((flt *) p);
  char *str =
    in("from_%d=1 & to_%d=1 & price<=%d",
        d.from, d.next, d.cash);
  Ticket t = parse_ticket(str);
  fly(d.next, d.to, d.cash - t.price);
  cm();
}
void fly(int from, int to, int cash) {
  flt_info    flt[N];
  func_ptr    choices[N];
  func_param  params[N];
  if (from == to) return;
  visited[from] = true;
  vector<int> next = unvisited_neighbors();
  for (size_t i = 0; i < next.size(); i++) {
    flt[i] = flt_info(from, next[i], to, cash);
    choices[i] = wait_for_ticket;
    params[i] = (func_param) &(flt[i]);
  }
  gcc(choices, params, next.size());
}
int main() { fly(0, 3, 100); return 0; }
```

Table 1 shows the result of the experiment. Both examples are repeated nine times, and the minimum and the maximum number of worlds given. We see that the number of worlds can be small even with speculation and the increase is also limited. Notice that we only obtained a maximum of 67 worlds, which is much smaller than the theoretical bound of 2^{20} worlds (10 pairs of Bob vs. Jill agents). The *response time* is the time for the coordination server to process the first GCC request. The *turnaround time* is the time to execute an agent program from start to finish, i.e. when all processes forked by this program exit. There is variation in times and number of worlds because GCC requests may be delayed randomly, and there is non-determinism in the execution of the agent programs/processes. For the maximum

		Bob-Jill cases		Flight cases	
		min	max	min	max
Max # of worlds		11	67	11	53
Max storage (MB)		40.6	53.5	17.8	89.8
Response time (s)	max	9.22	133.62	10.34	67.91
	avg.	4.44	44.48	3.36	17.05
Turnaround time (s)	max	9.23	1011.91	10.34	67.91
	avg.	5.07	318.19	3.36	17.05

Table 1: Experiment statistics

case of maximum turnaround time in the Bob-Jill cases, the higher latency is caused by the use of the opportunistic forking mechanism which delays requests according to the number of worlds. Thus, even when there is no more requests, delayed requests may still have to wait. The response times are not small enough because of the delays as well. This is a tradeoff between usage of system resources versus latency.

5. Conclusion

We have presented a concrete system architecture and associated implementation and optimizations for GCC. This paper shows that GCC can be practical and realistic. We show how to program in GCC by using our runtime system and how the agent programs work. The use of C/C++ as a host language demonstrates that GCC is language independent and embeddable in mainstream languages.

References

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

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

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

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

[5] H. P. Nii. *Blackboard Systems*. Addison Wesley, 1989.

[6] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[7] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *SLP*, pages 505–520, 1994.

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

[9] G. Smolka. The definition of kernel oz. In *Constraint Programming*, pages 251–292, 1994.

[10] G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards, 1994.