Generalized Committed Choice

Joxan Jaffar[1], Roland H.C. Yap[1] and Kenny Q. Zhu[2]

[1] School of Computing
National University of Singapore
{joxan,ryap}@comp.nus.edu.sg

[2] Department of Computer Science
Princeton University
kzhu@cs.princeton.edu

Abstract. We present a generalized committed choice construct for concurrent programs that interact with a shared store. The generalized committed choice (GCC) allows multiple computations from different alternatives to occur concurrently and later commit to one of them. GCC generalizes the traditional committed choice in Dijkstra's Guarded Command Language to handle *don't know* non-determinism and also allows for speculative computation. The main contribution of the paper is to introduce the GCC programming construct and the associated semantics framework for formalizing GCC. We give some experimental results which show that the power of GCC can be made practical.

1 Introduction

Nondeterminism means that a computation may need to choose between two or more choices [9]. *Don't care* non-determinism, or *committed choice*, is the most commonly used form of nondeterminism in concurrent programming systems, e.g. Occam [10] and Concurrent Prolog [17]. It's also the basis of many non-deterministic programming constructs such as guarded commands [5], CSP [9], and π-calculus [13]. The original form of committed choice is the *guarded command set* (Dijkstra's guard) [5], $G_1 \rightarrow S_1 \ \square \ G_2 \rightarrow S_2 \ \square \ \cdots \ \square \ G_n \rightarrow S_n$ where G_i is a logical expression, the *guard*, and S_i is a list of statements. The meaning of Dijkstra's guard is that one can choose any S_i to execute so long as its guard G_i is true. Otherwise if all guards are false, then it aborts. Thus the choice gives rise to a form of *don't care* non-determinism in contrast with the *don't know* non-determinism used in OR-parallel logic programming [7] which explores a search space non-deterministically to find solutions.

In a concurrent programming setting, a meaningful program contains operations that manipulate its environment allowing it to interact with other running programs. A key characteristic of Dijkstra's guard (also guards in Concurrent Logic/Constraint Programming [16]) is that guard G_i is meant as a test to choose an alternative to commit to before performing any other operations which modify the environment. We call this, *early committed choice*.

In this paper, we propose a new choice construct which allows the commit to occur anywhere within the choice. We call this, *generalized committed choice* (GCC), since it generalizes the idea of early committed choice. We assume that all processes are operating in a shared environment in which all variables are

global and shared (similar to [11]). We call the environment a *store*. The processes don't interact with each other directly, but instead communicate through modifying the values of the variables in the store. The only operations allowed are global variable assignments.

Next we will give a simple motivating example which explains why GCC is interesting along with some related work. The rest of the paper is organized as follows. Section 2 presents a small programming language which we embed GCC. Section 3 introduces the basic runtime structure for GCC. The operational semantics of GCC is given in Section 4. The meaning and alternatives of commit is discussed in section 5. Section 6 discusses how to make GCC practical and gives some experimental results.

1.1 A motivating example

The following example motivates the kinds of application of non-deterministic choice which are ideal applications for GCC. Imagine two people, among others, participate in an online automated second hand product trading system.

Bob is a photographer who wants to upgrade his equipment. He has two choices: either sell his old camera and buy a better one; or, keep the old camera but sell his old lens and buy a better lens. To avoid ending up with two cameras or selling all his equipment but unable to upgrade, only one scenario should occur. Jill wants to downgrade and either sell her good camera or her good lens. Using the proceeds from one of the above sales, she can now buy an average camera. To maximize buying and selling opportunities, we assume that the buying and selling of items can happen in any order.

We can program the requirements of Bob and Jill as follows. Exclusive choice is written as `XOR`. We assume that the market has a clearing function, which matches a buy action with a sell action, and vice versa. Thus both the buy and sell operations are synchronized and block if the corresponding action is not present.

```
 Bob:                                  Jill:
(buy(goodlens); sell(averagelens))    (sell(goodlens); buy(averagecam))
              XOR                                   XOR
(buy(goodcam); sell(averagecam))      (sell(goodcam); buy(averagecam))
```

It is easy to see that there is a perfect match between Bob and Jill as Bob can buy Jill's good camera and then sell Jill his average camera. Thus there is a way in which both parties can be satisfied. However, since each party is not directly aware of the other, in this setting, we want them to be able to act independently.

Bob could choose one of the following two non-speculative strategies. The first is for Bob to take a bet on one of the choices and commit to that choice. For example, choose on the first choice that gets to make some progress. That is, if a good lens is for sale, then buy the good lens and take a risk by waiting for a buyer for the average lens. This ignores the second possibility to buy a better camera. Such a "bet-and-risk-it" strategy has obvious pitfalls. What happens if one makes the wrong choice? In the example above, if Bob's program finds a match with Jill's choice of selling good lens first, then the lens (1^{st}) choice

of Bob will be committed and the camera (2^{nd}) choice will be eliminated. But as it turns out, Jill wants to buy an average camera and not an average lens after buying a good lens. As such, Bob and Jill are deadlocked, both waiting to complete their trade.

The second and more conservative strategy is for Bob to wait in both of his choices until the conditions for both buying and selling actions are met, and then do both actions atomically. While this is a safer option than the previous one, Bob will certainly miss the trading opportunity with Jill as Jill will not buy his average camera until she has sold her good camera. Both parties will be blocked even when there is a potential solution available.

Although our simple example has only two players, in a real life marketplace, much larger dependency cycles involving more parties may exist. These cyclic dependencies cannot be resolved through the use of the above early commit strategies. As we shall see next, the late commit in GCC solves this problem.

1.2 The Generalized Committed Choice Model

The generalized committed choice allows a new strategy which increases the probability of getting a solution. As Bob has two choices, to maximize his chances, he would like to be able to attempt both choices simultaneously and non-deterministically, and choose the one which succeeds. This leads to a form of speculative computation. We achieve this by having the computation in each choice operate in its own independent "world" containing an independent store. So one world does not effect the other. Now, when Jill comes to the system, her program will join the two existing worlds Bob's program has created. Since Jill also has two choices, her program will further split each world it is living in, and this creates four worlds altogether, each of which represents a possible interaction between Bob's and Jill's choices.

Here, everybody is given the full opportunity to complete their actions: while Bob may take Jill's good lens and get stuck in one world since he cannot sell his good lens; he may able to buy Jill's good camera and sell his average camera to her in another world and then eventually complete the transaction. Thus we also need a way of removing unwanted possibilities which represent other worlds and computations. In every world, Bob's and Jill's computation operate in their own independent reality. They can buy and sell items as if there is no speculation.

The model of GCC enables a programming paradigm where a computation can have a number of distinct possibilities. For simplicity, let us say there are two choices, α and β. We assume that the computations operate on a shared store which can change over time due to external events or through actions of a program. In a choice construct, we allow both α and β to proceed concurrently but isolated from each other. Although there is a form of isolation among choices within a program, an important property is that multiple programs (e.g. Bob and Jill's programs) interact with each other through different versions of the store, where each store represent one possibility. After some computation, one of the choices, say α can choose to commit. This has the effect as if the other possibility β never existed. When we have more than one user, all user choices

are multiplied to form a number of worlds. Allowing speculation means that computation results in multiple rather than one store/world.

This paper proposes a programming model for speculation in a new don't-know non-determinism and concurrency context. The central contribution is a new programming construct GCC which generalizes early committed choice. We formalize the complicated semantics of GCC. The main challenge of the semantics is to deal with the notion of commit in the context of multiple worlds. We demonstrate in some experiments that the expressive power can be made practical. In our producer-consumer experiments, the growth in the number of worlds, total size of the store in all the worlds, and the number of program instances executing can be contained.

1.3 Related work

A database transaction provides *atomicity* and *isolation* [15]. This can be used in the second strategy depicted earlier. The drawback is that it is unlikely to give the desired result since that means all blocking conditions are satisfied at once. Also, relational databases and SQL do not handle non-determinism, so Bob and Jill cannot specify their choices.

Long-lived transactions such as "Sagas" [6] do away with the isolation property of transactions. Partial changes to the database are visible to other transactions. Only one level of nested transactions is allowed. Operations in Saga can be unsafe, i.e. if one process cannot go through, then the whole saga needs to be compensated. Since there is concurrency, sometimes no compensation operations are possible, and hence the system becomes irreparably inconsistent. Saga does not handle our example as it does not provide non-deterministic choice. Even if Saga is extended, multiple choices are operating in the same environment, which means once Bob has bought the good lens for example, he will not have the money to pay for good camera, even if it is available. Furthermore, the compensating transactions have to be provided by the user program rather than being resolved by the system.

Dijkstra's guards [5, 14] and concurrent constraint programming (CCP) [16] techniques such as GHC [19] and Oz [18] use *early committed choice* to handle non-determinism. Early committed choice can be used to implement the *bet-and-risk-it* strategy, but as we have shown, this may lead to a blocked/deadlocked computation. Furthermore, CCP languages require monotonic stores and is thus not applicable in our context. Perhaps the closest relative to GCC is the *deep guards* in GHC and some other CCP languages, where recursion is allowed in the guards. However, although commit can be postponed with deep guards only reads and no writes are allowed in the guards and hence the store is not changed. This is in contrast with the GCC model where updates to the store are permitted in the choices before commit.

Transaction Logic ($\mathcal{TR}$) [1] is an algebra that offers a logical framework to model traditional database transactions. It supports the sequence of transactions (with updates) through the $\otimes$ operator (similar to our ";") and non-determinism through the $\oplus$ operator. The non-determinism here is equivalent to

an early-committed choice. $\mathcal{TR}$, however, does not support long-lived transactions. Concurrent Transaction Logic ($\mathcal{CTR}$) [2] adds the concurrent conjunction to $\mathcal{TR}$, which gives parallel interleaving of different sequences. It is different from our notion of late choice where each choice runs in isolation.

In composable memory transactions (CMT), the `orElse` construct [8] provides a way to state several *alternatives* in memory transactions. The transaction `s1 ‘orElse‘ s2` first runs `s1`; if it blocks (retries), then `s1` is abandoned with no effect, `s2` is run. If `s2` is also blocked (retries), then the whole transaction retries. In a way, the `orElse` construct offers a don't know type of non-determinism as it attempts the choices one by one until a satisfying one is found. The semantics of `orElse` differs from GCC, in the former the choices are attempted sequentially, while GCC allows all choices in parallel. In addition, CMT requires isolation of transactions, whereas GCC allows interactions between the programs during the resolution of don't-know choices. Hence, CMT cannot be used to code our Bob/Jill example directly.

In more recent development, transactional events [4] introduced a choice construct `chooseEvt` in a synchronous message passing setting. This construct is similar to the `orElse` in that there is no isolation between the choices, though the two choices are executed concurrently. Therefore it is not able to solve Bob and Jill's problem, either. Moreover, transactional event does not offer an explicit commit operation, therefore `chooseEvt` is not able to commit to a branch before the entire branch has been successfully completed, whereas GCC programs can commit a branch anywhere a commit point is reached.

The *Event-Condition-Action* rules in active databases [20, 3] provide some degree of reactivity to the users, it is still early committed choice. In addition, the actions carried out in ECA rules are either simple database read/write operations or user-defined procedure calls. But in all cases, these actions are done atomically and in isolation, hence interleavings are not possible.

2 Programming with GCC

We illustrate the programming paradigm of GCC with the following simple setting. There are a number concurrent or parallel programs interacting with a common runtime system, providing a global computation environment or a global memory we call a *store*. Without loss of generality, we assume programs do not use any local variables. Synchronization is achieved by use of the common store and the use of a blocking guarded action. For now, we simply assume the store is a piece of shared memory which contains variable-value mappings. The main operation on the store are variable assignments which are atomic.

We now present simple programming constructs for programming with GCC. Although we have chosen a simple setting, it should be clear that GCC can be easily integrated into more complex programming languages and concurrent/parallel systems.

We introduce the a minimal concurrent language with GCC for programs r defined as follows:

r	::=	`noop`	no operation
		$x := v$	atomic assignment
		`if` c `then` r_1 `else` r_2	conditional
		`while` c `do` r_1	loop
		$c \Rightarrow \delta$	guarded atomic action
		$r_1;\ r_2$	sequence
		$r_1 \oplus r_2$	GCC
		`cm`	commit this choice
		`cu`	commit other choice

The first four constructs are rather standard and thus require little explanation. The assignment operation assigns a value v to a global variable x in the store atomically. Boolean condition c is tested in both the conditional, loop and guard constructs. An example of c is $x + y \leq 10$.

Guarded atomic action, or guard in short, is provided to allow for reactive behavior and enable synchronization among the programs. $c \Rightarrow \delta$, blocks until condition c is true w.r.t. the store and then atomically executes δ. δ is an action such as noop, assignment, `cm` and `cu`, all w.r.t. the store. A guarded sequence of actions can be decomposed as:

$$c \Rightarrow (\delta_1;\ \delta_2; \ldots;\ \delta_n) \equiv c \Rightarrow \delta_1;\ c \Rightarrow \delta_2; \ldots;\ c \Rightarrow \delta_n.$$

The choice construct, $r_1 \oplus r_2$, defines two computations r_1 and r_2 which are to be executed speculatively. For simplicity, we only deal with binary choices, and it is straightforward to extend to an arbitrary number of choices. Both r_1 and r_2 execute concurrently with any updates isolated from each other. Nested choices are allowed. Unless otherwise stated, in the remainder of this paper, we refer to generalized committed choice simply as *choice*.

Within a choice, there are two special operations, namely `cm` and `cu`, which stand for "commit me" and "commit you". These can only be used within the scope of a choice where they refer to the *innermost* enclosing choice structure. `cm` expresses the intention to *commit* to this branch of a choice and to remove the other branch as if it didn't exist; `cu` expresses the intention to *terminate* this branch of a choice and commit to the other branch. Notice that `cm` and `cu` are *not* symmetrical since after executing `cm` in a branch, the program can continue, whereas in the case of `cu`; the program instance is *halted*. If `cm` and `cu` are used outside the scope of a choice, they have no effect. Furthermore, only the first use of `cm` or `cu` has any effect within a choice branch. For example, in this program,

$$(r_1 \oplus_1 ((r_2; \texttt{cm}; \texttt{cm}) \oplus_2 r_3)),$$

only the first `cm` after r_2 is effective and refers to the left branch of choice $\oplus_2$. The second `cm` is ignored, and it is *not* referring to the right branch of choice $\oplus_1$. Note we numbered the choice structures by subscripts just for the succinctness of explanation.

Static scoping is used for simplicity with `cm` and `cu`. Furthermore, we require that every choice branch must have a commit operation, so that a choice must eventually be "committed". This can be achieved by systematically adding `cm` to the end of every choice.

We remark that we have not defined a parallel composition construct here because our setting is of an open system where independent programs running in parallel can be introduced from the outside. One could, for example, implement parallel composition as the birth of a new program (see Section 4.1). Thus, one could add "fork-like" constructs to the language. We haven't done so for the interest of simplicity.

The Bob example can be re-written below in our simple programming language as follows:

```
(goodlens ≥ 1 ⇒                          (goodcam ≥ 1 ⇒
      goodlens := goodlens - 1;      ⊕         goodcam := goodcam - 1;
averagelens := averagelens + 1); cm     averagecam := averagecam + 1); cm
```

For simplicity, we have not dealt with the details of the transactions and have only expressed the requirements of the example as availabilities of the items.

Time variables are useful to write conditions involving external time. For example, this can be used to express *timeouts*. Consider a program fragment r which is to be executed as long as the timeout hasn't expired. This is expressed as, $r \ \oplus ((time \geq t) \Rightarrow \texttt{cm})$, so that when time has reached t, the right branch can commit and the effect is that r is aborted.

3 Worlds and Multi-Worlds

We present informally a suitable runtime structure for GCC. We define a world, a store and programs and extend the definitions to multi-worlds.

3.1 World

The basic computation space for GCC is a *world*, and is denoted by w_i, where i is a unique identifier for a world. One can think of a world as a shared memory computer running multiple processes. Associated with a world is its memory, we call this a *store*, and is denoted by Δ. The store contains, in its simplest form, a set of variable-value mappings, such as $\{x = 2, y = -1, z = 0\}$.

Every world executes a dynamic number of processes or programs in an interleaving model of concurrency. New programs may enter the world by *birth*, or depart the world by *death*, or completion. Because of such dynamism, we say the world is *open*. Programs execute when the system picks one continuation from the set, and advances it by executing an instruction, known as a program step. Initially, *cids* is empty.

We define a *world* as a pair (Δ, Σ), where Δ is a store and Σ is a set of continuations of all the programs interacting with Δ. We denote the store as a triangle ($\triangle$), annotated with an id of the world (and also the store itself). The set of continuations in that world is denoted by Σ below the triangle. Fig. 1 shows some worlds.

3.2 Multi-world

The runtime structure of GCC in general consists of a collection of worlds organized in a tree structure. We call this structure a *multi-world*. The leaves of the tree are worlds, and the intermediate nodes are choice nodes $\oplus_{id}$, where id is a unique identifier. The multi-world is used to represent different possible runtime scenarios for programs to interact. Program continuations associated with one world are executed in isolation from that of other worlds.

The operational semantics of the GCC and the programming language is centered around the creation, evolution and deletion of a number of worlds.

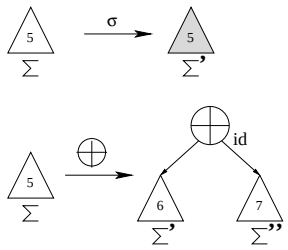


Fig. 1. Evolution of a world to a multi-world

Given a world, the execution of one program step by a continuation either evolves the world to a new state (store + continuations), or it can split the world into two new worlds using a choice, i.e. a choice point. This is illustrated in Fig. 1. The upper transition is when a continuation issues an update from a continuation σ, e.g. x := 10, which changes store Δ_5 to its new state (denoted by the gray color) and updated continuations Σ'. The lower transition shows the splitting of one world into two, when a continuation issues a choice $\oplus$. This creates two new stores, Δ_6 and Δ_7, each of which is a copy of the original store Δ_5, and also two sets of continuations Σ' and Σ'' for each branch in the choice point. Later, we show how `cm` and `cu` help reduce the number of worlds by "chopping" sub-trees from the multi-world. We can think of $\oplus_{id}$ as a logical XOR.

4 Operational Semantics

We now describe the operation semantics of GCC and the simple programming language using state transitions on multi-worlds.

Let Δ be a store or a database which is a finite mapping between an infinite set of variables and values: $\Delta : x \mapsto v$. We write $\Delta[x \mapsto v]$ to denote the change of the value of x in Δ to v.

Let p be a program, pc be the program counter and $cids$ be a sequence of choice id's where each choice id is a unique number. An instance of a program,

or a *continuation* σ is a triple: $\langle p,\ pc,\ cids\rangle$. Let $next(p, pc)$ be the pc of the next instruction, $next_l(p, pc)$ be the next instruction after σ which corresponds to the left branch of a choice, and $next_r(p, pc)$ be the next instruction which corresponds to the right branch of a choice.

Let w be a world, we recursively define a multi-world $\mathcal{W}$ to be:

$$\mathcal{W} = w \mid \mathcal{W}_1 \oplus_{id} \mathcal{W}_2$$

We can treat a multi-world as a tree, and a world as a leaf of the tree. Given a multi-world $\mathcal{W}$, the function $worlds(\mathcal{W})$ returns all worlds (or leaves) of $\mathcal{W}$.

We define δ to be a function that maps a world to a store, i.e. $\delta : w \mapsto \Delta$, and define θ to be a function that maps a world to a set of continuations currently associated with that world, i.e. $\theta : w \mapsto \Sigma$, where Σ is a set of continuations.

The state of a GCC system is a triple, $(\mathcal{W}, \delta, \theta)$, where $\mathcal{W}$ is a multi-world, and δ and θ are functions defined above. The system evolves by updating some or all of these parameters.

Let $view(\Delta) = \{(x = v) \mid \Delta(x) = v\}$, i.e. the view of a store is a set of all variable-value equations induced by the store. We further define the *conjunctive view* and *disjunctive view* of a multi-world as follows:

$$\begin{aligned}
\mathcal{CV}(w) &= view(\delta(w)) \\
\mathcal{CV}(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2) &= \{(x = v) | (x = v) \in \mathcal{CV}(\mathcal{W}_1) \wedge \\
&\quad (x = v) \in \mathcal{CV}(\mathcal{W}_2)\} \\
\mathcal{DV}(w) &= view(\delta(w)) \\
\mathcal{DV}(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2) &= \{(x = v) | (x = v) \in \mathcal{DV}(\mathcal{W}_1) \vee \\
&\quad (x = v) \in \mathcal{DV}(\mathcal{W}_2)\}
\end{aligned}$$

The conjunctive view of a multi-world represents the variables which hold identical values across all worlds in this multi-world; the disjunctive view gives a set of all possible values of all the variables in these worlds.

4.1 The GCC System

A GCC system is triple $(\mathcal{W},\ \delta,\ \theta)$. In general, the GCC system advances by the following transition rule:

$$(\mathcal{W}, \delta, \theta) \longrightarrow (\mathcal{W}', \delta', \theta') \tag{1}$$

At each time step, the system non-deterministically chooses to take either a program step or a system step. To take a program step, the system picks a continuation in one of the worlds and executes its current instruction by one of the following program step rules: assignment, test, choice or commit. To take a system step, the system evaluates certain conditions of the multi-world, and makes changes such as pruning the multi-world tree, adding new programs or deleting completed programs from the system by selecting a suitable system step rule. The evolution of multi-worlds is that it world grows due to program steps; and it shrinks due to system steps. We use '+' below to mean set union, '−' to mean set difference where appropriate, and '∗' to mean any value (wildcard).

Program Step Rules We now define the transitions of (1) attributed to a user program. In what follows, we shall be selecting a continuation $\sigma = \langle p, pc, cids\rangle$ from $\mathcal{W}$, and describe its one-step execution. We write w to denote the world associated with σ, and Δ to denote the corresponding store $\delta(w)$. We write pc' to denote $next(p, pc)$ except for test or choice. For a test, we write pc_1 and pc_2 for the next pc after the test. For a choice, we write pc_l and pc_r for the next pc advancing to the left and the right branch in a choice.

We now proceed by case analysis on the construct corresponding to the current program point $p[pc]$:

Assignment $x := v$
$\Delta' = \Delta[x \mapsto v]$, $\delta' = \delta[w \mapsto \Delta']$, and
$\theta' = \theta[w \mapsto (\theta(w) - \{\sigma\} + \{\sigma'\})]$ where $\sigma' = \langle p, pc', cids\rangle$.

Test or Guard $c?$
$\theta' = \theta[w \mapsto (\theta(w) - \{\sigma\} + \{\sigma'\})]$
where $\sigma' = \langle p, pc_1, cids\rangle$ if $view(\delta(w)) \models c$;
$\sigma' = \langle p, pc_2, cids\rangle$ if $view(\delta(w)) \not\models c$.

Choice $\oplus$
$\mathcal{W}' = \mathcal{W}[w \leftarrow w' \oplus_{id} w'']$,
$id = \langle p, pc, s\rangle$ where s is a unique number,
$\delta' = \delta - \{w \mapsto \delta(w)\} + \{w' \mapsto \delta(w)\} + \{w'' \mapsto \delta(w)\}$, and $\theta' = \theta - \{w \mapsto \theta(w)\} + \{w' \mapsto (\theta(w) - \{\sigma\} + \{\sigma_l\})\} + \{w'' \mapsto (\theta(w) - \{\sigma\} + \{\sigma_r\})\}$
where $\sigma_l = \langle p, pc_l, cids.append(id)\rangle$ and $\sigma_r = \langle p, pc_r, cids.append(id)\rangle$.

Commit `cm`
$\delta' = \delta[w \mapsto (\delta(w) \cup \{\mathtt{cm}_{id} \mapsto 1\})]$ and $\theta' = \theta[w \mapsto \theta(w) - \{\sigma\} + \{\sigma'\}]$
where $id = cids.last()$ and $\sigma' = \langle p, pc', cids.droplast()\rangle$.

Commit `cu`
$\delta' = \delta[w \mapsto (\delta(w) \cup \{\mathtt{cu}_{id} \mapsto 1\})]$ and $\theta' = \theta[w \mapsto \theta(w) - \{\sigma\} + \{\sigma'\}]$
where $id = cids.last()$ and $\sigma' = \langle p, pc', cids.droplast()\rangle$.

In an assignment step coming from a continuation of a world w, the store of that world is updated and the continuation steps forward.

In a test step, no store is changed, except the continuation being activated advances to a new *pc*. In case of a guard, that *pc* is not changed if the test failed (which can be thought of as a busy loop [1]); in case of conditionals or while loop, the *pc* changes to different values depending on the result of the test.

If a continuation σ issues a choice in world w, that world is split into a multi-world $w' \oplus_{id} w''$, with the store of w copied to w' and w'', and the σ of w' advances to the left branch while the σ of w'' advances to the right branch. A new choice id is dynamically generated.

The rules for `cm` and `cu` simply updates the affected world's store with a new variable and value: $\mathtt{cm}_{id} = 1$ or $\mathtt{cu}_{id} = 1$ to indicate that a previously issued choice has been committed on one of the branches.

[1] In practice, a triggering mechanism which indexes all the blocked guards and only fires the ones which are enabled can be used [11, 12].

System Step Rules We now complete the definition of (1), this time by defining transitions attributed to the system.

Birth $(\mathcal{W}, \delta, \theta) \xrightarrow{b(q)} (\mathcal{W}, \delta, \theta')$

where q is a new program, and $\theta' = \theta[w \mapsto \theta(w) + \langle q, 0, []\rangle]$ for all $w \in worlds(\mathcal{W})$.

Death $(\mathcal{W}, \delta, \theta) \xrightarrow{d(q)} (\mathcal{W}, \delta, \theta')$

where for all continuations $\langle q, pc, *\rangle$ in $\mathcal{W}$, $q[pc] =$`end`, and
$\theta' = \theta[w \mapsto (\theta(w) - \{\langle q[pc] = \texttt{end}, *\rangle\})]$ for all $w \in worlds(\mathcal{W})$.

Pruning

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_r} (\mathcal{W}_1, \delta', \theta') \tag{2}$$

where $\mathcal{CV}(\mathcal{W}_1) \models cm_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_2)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_2)\}$.

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_l} (\mathcal{W}_2, \delta', \theta') \tag{3}$$

where $\mathcal{CV}(\mathcal{W}_2) \models cm_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_1)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_1)\}$.

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_l} (\mathcal{W}_2, \delta', \theta') \tag{4}$$

where $\mathcal{CV}(\mathcal{W}_1) \models cu_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_1)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_1)\}$.

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_r} (\mathcal{W}_1, \delta', \theta') \tag{5}$$

where $\mathcal{CV}(\mathcal{W}_2) \models cu_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_2)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_2)\}$.

The birth and death of programs formalize the launch and completion of an agent program. The birth of a new program q, denoted by $b(q)$, adds instances of this program to every world in the multi-world. For simplicity, we treat every program as being unique even when they are identical. The only way programs are copied is through the execution of choice (see section 4.1). The death of a program q is happens when it has reached the end of the program in every instance of its appearance in the multi-world, then all its instances are removed from the multi-world.

There are 4 pruning rules, two due to `cm`, and two due to `cu`. Rules (2) and (3) chop off the right (left) subtree rooted at $\oplus_{id}$, if the conjunctive view of the left (right) subtree has implied that $\texttt{cm}_{id} = 1$. Symmetrically, Rules (4) and (5) check if the left (right) subtree of $\oplus_{id}$ has a conjunctive view that implies

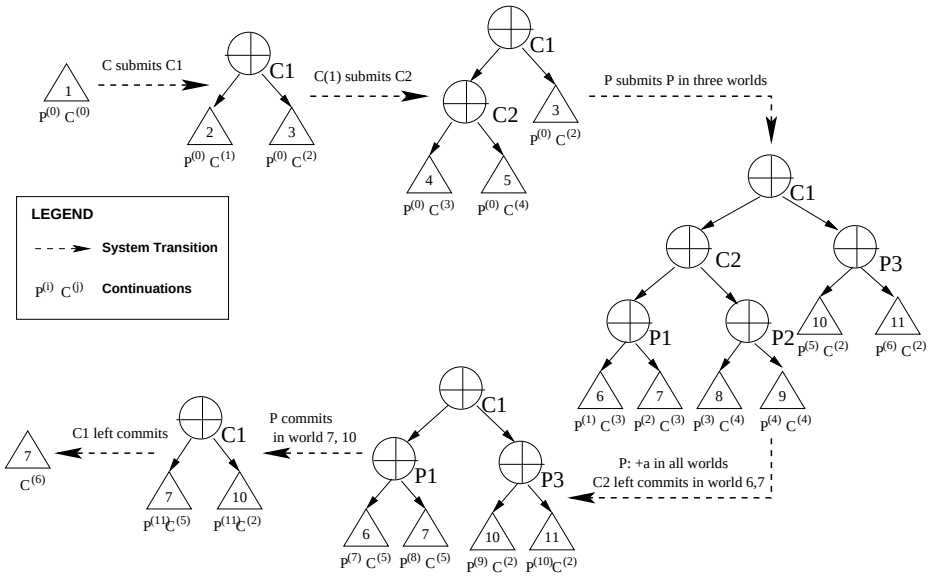
$\mathtt{cu}_{id} = 1$. In other words, if a choice identified by id has committed to its left branch in all the worlds in the left sub-tree of a multi-world rooted at $\oplus_{id}$, then all other worlds under $\oplus_{id}$ in which the right branch of choice id is attempted are discarded. If the choice has given up (commit you) left branch in all worlds of the left sub-tree of the multi-world rooted at $\oplus_{id}$, then all worlds in the left sub-tree are discarded. Note that there can be more than one multi-worlds rooted at $\oplus_{id}$, but they are treated separately under the pruning rules.

4.2 An example of multi-world transitions

Consider the following producer/consumer example.

$$P ::= (a := a + 1;\ b := b + 1;\ \mathtt{cm}) \oplus (a := a + 1;\ c := c + 1;\ \mathtt{cm})$$

$$C ::= (((a \geq 1 \Rightarrow a := a - 1;\ \mathtt{cm})\ \oplus_2 (b \geq 1 \Rightarrow b := b - 1;\ \mathtt{cm}));\ \mathtt{cm}) \oplus_1 ((time \geq 10) \Rightarrow \mathtt{cm})$$



Continuations Listing:

$P^{(0)}$: $\langle P,\ 0,\ [] \rangle$	$P^{(1)}$: $\langle P,\ 1,\ [P1] \rangle$	$P^{(2)}$: $\langle P,\ 5,\ [P1] \rangle$	$P^{(3)}$: $\langle P,\ 1,\ [P2] \rangle$
$P^{(4)}$: $\langle P,\ 5,\ [P2] \rangle$	$P^{(5)}$: $\langle P,\ 1,\ [P3] \rangle$	$P^{(6)}$: $\langle P,\ 5,\ [P3] \rangle$	$P^{(7)}$: $\langle P,\ 2,\ [P1] \rangle$
$P^{(8)}$: $\langle P,\ 6,\ [P1] \rangle$	$P^{(9)}$: $\langle P,\ 2,\ [P3] \rangle$	$P^{(10)}$: $\langle P,\ 6,\ [P3] \rangle$	$P^{(11)}$: $\langle P,\ 9,\ [] \rangle$
$C^{(0)}$: $\langle C,\ 0,\ [] \rangle$	$C^{(1)}$: $\langle C,\ 1,\ [C1] \rangle$	$C^{(2)}$: $\langle C,\ 10,\ [C1] \rangle$	$C^{(3)}$: $\langle C,\ 2,\ [C2,\ C1] \rangle$
$C^{(4)}$: $\langle C,\ 5,\ [C2,\ C1] \rangle$	$C^{(5)}$: $\langle C,\ 8,\ [C1] \rangle$	$C^{(6)}$: $\langle C,\ 12,\ [] \rangle$	

Fig. 2. A worked example

Fig. 2 illustrates the dynamic evolution of the multi-world with the two concurrent programs P and C. The initial store is $\Delta_1 = \{a = 0, b = 0, c = 0, time = 0\}$. We number the choice nodes of P and C as $P1, P2, P3, C1, C2$. $P^{(i)}$ and $C^{(i)}$ denote the continuations with the details given at the bottom.

Suppose C gets to make its choices first, the result is three worlds with the two choice nodes $C1$ and $C2$ in the multi-world. Then P starts to issue a choice which multiplies to six different worlds. As P produces a and b in one branch and a and c in another, before P commits in any of its branches, choice $C2$'s left branch can consume a and commit. The commit adds the information $\texttt{cm}_{C2} = 1$ to both worlds, hence the prune step can be applied to prune off the worlds of w_8 and w_9. In the $P3$ subtree, P commits in w_{10} first and deletes w_{11}. P can also commit in the the right branch of the $P1$ subtree, using a prune step to prune off the left subtree of $P1$, namely w_6. At this point, there are only two worlds left, w_7 and w_{10}. Since P now has committed in all worlds (w_7 and w_{10}), it exits from the system. Finally, the left branch of $C1$ commits, which kills world w_{10} so there is only one remaining world w_7.

5 On the semantics of commit

The key issue of commit is when a `cm` or `cu` is executed in some world, which other worlds should be deleted and when to delete them. For our discussion, we use one of the multi-words from Fig. 2 as Fig. 3. Fig. 3 shows that the same syntactic choice in P can be issued multiple times in different parts of the multi-world. These choice nodes have different id's, as they are created in different worlds and hence belong to different *scopes*. Subsequently other programs executing in worlds w_6 to w_{11} can also issue choices, further expanding the tree.

The *coordinated commit* works as follows. A commit operation not only has to match syntactically with the choice structure it belongs to (which is identified by the program code p and the program counter pc, but also maps itself to the scope in which the choice was first launched into. For example, if a commit is executed in one of the worlds under node $P1$ in Fig. 3, then only some of the worlds under $P1$ should be deleted and not worlds under $P2$ or $P3$ in the tree. This is accomplished by using the choice id sequence, *cids*, in each program continuation. A commit always uses the id of the inner-most choice construct that surrounds this commit operation. Since the id's are generated dynamically as choices are issued, the same choice construct will obtain different id's in different scopes. Thus the matching of commits with the correct choice nodes is done automatically.

We now discuss other alternatives for the semantics of commit, and argue why the given semantics in section 4 is selected. We will illustrate these semantics in Fig. 4, 5 and 6. For simplicity, the continuations associated with the worlds are omitted. Where commit has been executed to a store Δ_i, we write $\texttt{cm}_{id}$ under Δ_i. The stores with the commits of interest are highlighted.

Absolutely eager commit prunes in an eager fashion. If a `cm` whose id is id is reached in any world, this world is committed and all other worlds within the

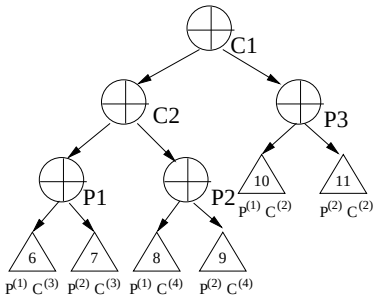


Fig. 3. A multi-world

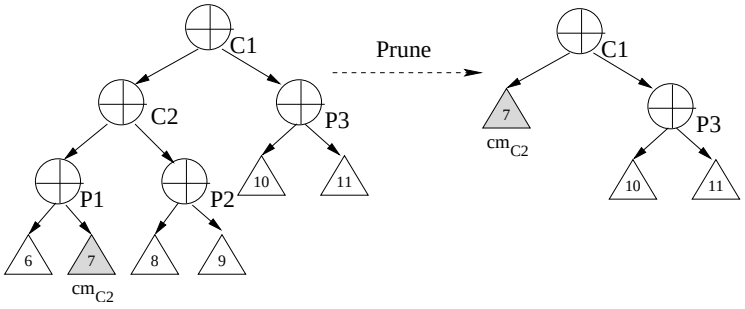


Fig. 4. Absolutely eager commit

scope of this `cm`, that is, under the subtree rooted at $\oplus_{id}$, are killed immediately, leaving just one world under $\oplus_{id}$. That effectively removed all the intermediate nodes including $\oplus_{id}$ in that subtree (see Fig. 4).

This form of commit does not seem very useful since it allows for minimal inter-play of choices, and the chances of achieving a useful speculation is small since other possibilities are eliminated immediately. In addition, with this semantics, `cu` does not make sense as `cu` may refer to a choice currently in many worlds and the system does not know which world to commit to.

A "less eager" kind of commit is *eager coordinated* commit. For a program $r ::= r_1 \oplus r_2$, if `cm` is reached in one of the worlds where r_1 is executed, then all worlds associated with r_2 are deleted immediately (see Fig. 5). The idea here is that syntactically r_2 is not a viable choice any more. Conversely, if `cu` is reached by r_2 in any world, then all worlds associated with r_2 are deleted immediately. However, in either cases, r returns only after r_1 has committed in *all* the remaining worlds.

This type of commit is "eager" because commit in one world kills the alternative choice in all other worlds; it is "coordinated" as the program only returns after the choice can commit in all remaining worlds. The following example of

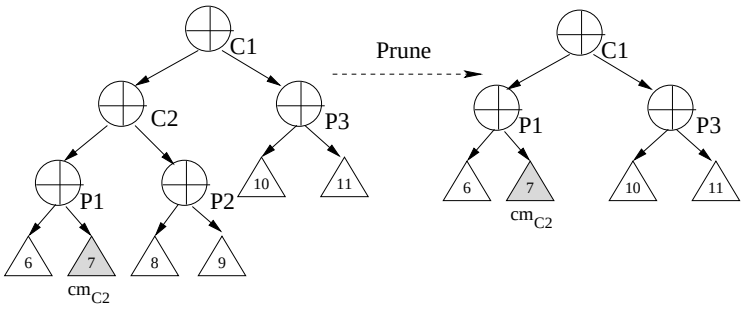


Fig. 5. Eager coordinated commit

two programs P and C shows a drawback of this semantics,

$$P ::= (+a; \texttt{cm}) \oplus (+b; \texttt{cm})$$
$$C ::= (?a \Rightarrow -a; \texttt{cm}) \oplus (?b \Rightarrow -b; \texttt{cm})$$

Four worlds are created with these two programs. We denote the world in which the left branch of P and left branch of C interact as $P_l C_l$ and likewise for other worlds. Assume nothing exists in the store initially, P produces a and b in all four worlds but hasn't committed. Now suppose C consumes a in $P_l C_l$ and commits, which kills worlds $P_l C_r$ and $P_r C_r$. However, as it turns out, P now commits in world $P_r C_l$ first and kills $P_l C_l$, which renders C in a blocked state. Had C not killed $P_l C_r$ and $P_r C_r$ early, both P and C may commit in world $P_r C_r$.

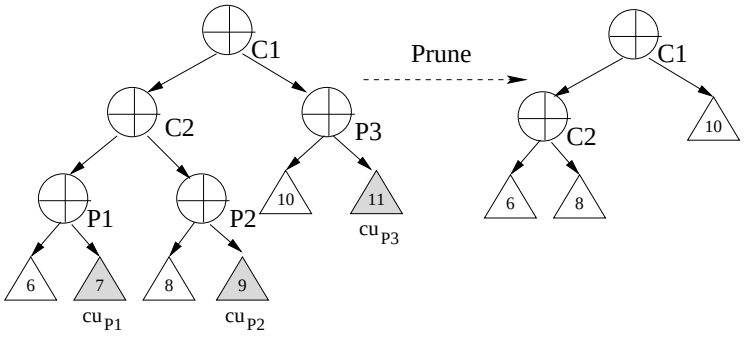


Fig. 6. Late coordinated commit

A more lazy form of commit is the *late coordinated* commit. Here the system only starts removing worlds when a program $r ::= r_1 \oplus r_2$ with a choice construct has committed its r_1 (or r_2) branch in *all* the worlds with which the r_1 (or r_2) branch continuations are associated. In other words, commits are not only

coordinated within a scope, but also across all scopes this program was associated with. For example, in Fig. 6, worlds 7, 9 and 11 are only removed when a `cu` in program P has been executed in all these worlds. Note that worlds 7, 9, 11 are the only worlds in which the right branch of P exists.

While this semantics increases the possibility of getting a solution and decreases the chances of deadlock from a system point of view, it is unduly conservative in when to prune the tree which may cause the multi-world to be too large. One drawback is when one branch continuation of r cannot proceed to commit due to blocking, no worlds due to r can be removed from the multi-world.

The coordinated commit introduced in Section 4 can be considered as a compromise between the eager coordinated commit and the late coordinate commit. It does not kill the alternative choice until `cm` of this choice has been reached in all worlds in the scope of the choice construct. This solves the deadlock problem caused by eager coordinated commit, depicted in the example above. At the same time, it does not over-delay the removal of the worlds so that the size of the multi-world can be contained more effectively.

Finally, we remark that it is possible to have a mixed semantics for commit. Though we have presented one fixed commit semantics in Section 4, it is straightforward to extend it to several possible semantics. In a closed system, concurrently interacting programs can be determined and controlled, hence it makes sense for programs to specify their desired version of commit. In this paper, we consider the more general setting of an open system where agents can dynamically submit arbitrary programs. Here, having multiple commit semantics makes less sense because unpredictable behavior of the system can negate the extra program control associated with the mixed semantics. For example, a late committed choice could be killed by an eager commit issued elsewhere.

6 Empirics

The late choice in GCC allows for more kinds of reactive interactions which involve speculation. This extra expressive power naturally comes with a price. When there are choices, this means that the choices in one program are multiplied with choices from other programs. This means that when choices do not or cannot commit for a long time, the multi-world can grow exponentially large. More worlds in the multi-world also mean more concurrency since there are more continuations. Furthermore, the space for the store might also increase with more worlds. We have developed the following implementation ideas which address the space and computation requirements.

Reducing storage — We define a *system root* to be the largest multi-world in the run-time system. Then a *differential view*, $\mathcal{DFV}$, of a non-system-root multi-world $\mathcal{W}_i \oplus_{id} \mathcal{W}_j$ is, $\mathcal{DFV}(\mathcal{W}_i) = \mathcal{CV}(\mathcal{W}_i) - \mathcal{CV}(\mathcal{W}_i \oplus_{id} \mathcal{W}_j)$. And for a system root $\mathcal{W}_0$, $\mathcal{DFV}(\mathcal{W}_0) = \mathcal{CV}(\mathcal{W}_0)$. This optimization re-organizes the multi-world so that, instead of having the data at the leaves, portions of the stores can be materialized in the internal nodes in the tree by using a differential view. In essence, this optimization stores common data as high as possible in the tree, to

reduce storage redundancy. One can use a strategy that periodically materializes the $\mathcal{DFV}$ at respective nodes. This makes pruning more efficient as the common view can be assessed higher in the tree, without going to the leaves to compute the conjunction. For efficient evaluation of guard conditions, we can make use of materialized disjunctive view $\mathcal{DV}$ at internal nodes. $\mathcal{DV}$ can be used as an indexing condition to approximate whether a change in the view might wake up a blocked guard. However, disjunctive views are large and can be too expensive to materialize. We instead store a *common property*, $\mathcal{CP}$, of that view $\mathcal{DV}$, such that $\mathcal{DV} \models \mathcal{CP}$.

Reducing the number of continuations — Under some safety conditions, we can collect continuations which are identical copies (due to other programs splitting the worlds) from the leaves, treat them as one copy, which we call the *synchronous continuation* and execute it at a higher node in the multi-world tree. In other words, instead of running many copies of the same program on the leaves, we run just one copy in a higher internal node, and thus save computation. For example, in program

$$p ::= \texttt{while}(w > 0)\ \texttt{do}\ w := w - 1$$

if variable w is currently not speculated (i.e. has one value) and will not be updated by any other programs in future, then executing p at a node higher in the tree has the same effect as executing multiple instances of p at the leaves of the sub-tree.

Reducing the number of worlds — We keep sub-trees generated by each independent program in a "chain" form so long as the programs don't have data dependency on each other. If and when a data dependency is required, the linear tree can be expanded partially and on demand. The following property describes an ideal case of structure sharing.

Property 1. Given a set of programs R whose data requirements are *disjoint* from each other, the structure-shared GCC runtime structure is a linear ordering of $|R|$ subtrees, where each subtree represents the runtime structure of a respective program.

To investigate the implementation tradeoffs in applications with speculation, we experimented with simulations based on the producer and consumer problem. For simplicity, there are n_t types of resources being produced and consumed. The producer programs do not involve a choice, but just produce up to 3 items of the same type at a time. The consumer programs have one GCC choice, where each branch attempts to consume two items of the same type provided they are available. The consumer blocks until the items become available.

This simple setup replicates a basic model of an economy where agents can bundle long transactions and speculate on different possibilities. It is similar to what the wish list in real time transactions such as booking a holiday, i.e. buying the appropriate combination of air tickets, hotel rooms, ground transportation, etc. Different choices arise due to different costs, routing possibilities, availability of the item, etc. For example, a different flight routing might necessitate a hotel room.

In the experiment, computations are executed by clock ticks. At each clock tick, either a producer program or a consumer program or no program is launched. At the same tick, every world in the system advances one step by picking one program (either producer or consumer) attached in this world and executes an instruction.

We identify two dimensions of simulation metrics:

- the level of overlapping interest by the producers and the consumers: high overlapping (HO) where $n_t = 5$ and low overlapping (LO) where $n_t = 100$.
- the relative rate of production against the consumption high production (HP), low(LP) and balanced production (BP)

We used six datasets (combinations of the above two dimensions, i.e. HO/HP, HO/BP, ...) with 50 producers and consumers randomly launched to the system with a random delay from 0 to 9 ticks between two consecutive program submissions. Due to lack of space, we only show two of the experiments, which are representative of the good cases where GCC does not incur much additional cost and the bad cases where GCC does lead to more overhead. Fig. 7 and Fig. 8 shows the resulting traces of three factors, i.e. total number of worlds in the multi-world, total number of program continuations and total number of data items stored, over a period of 500 time steps.

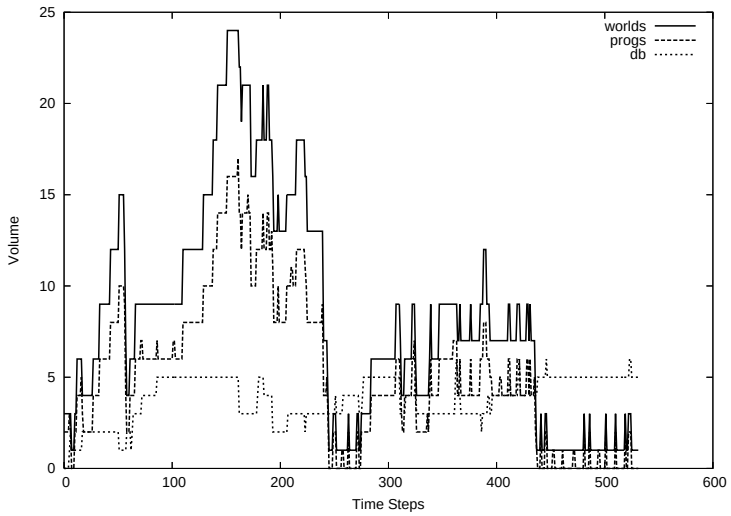


Fig. 7. Trace plot for HO/HP

We can see that the optimizations we described are successful in controlling the size of the multi-worlds and the storage requirements. For high overlapping cases such as Fig. 7, the data storage is consistently low, and all programs run to completion within 500 ticks given high and balanced production. In other

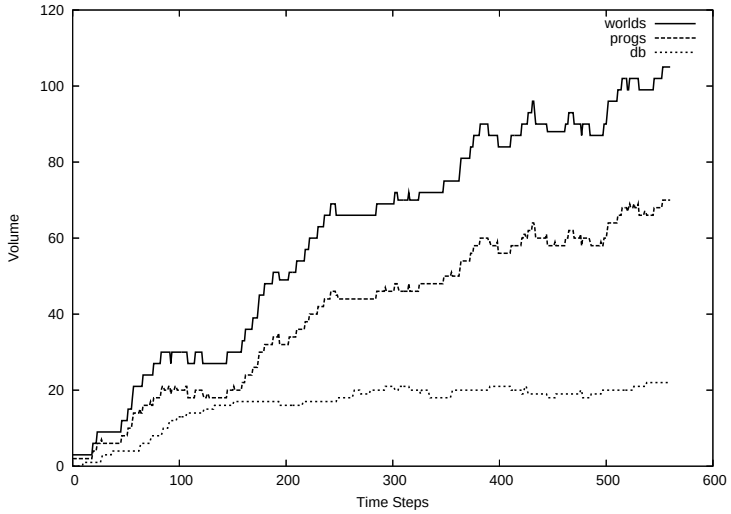


Fig. 8. Trace plot for LO/LP

more difficult cases like LO/LP in fig. 8, some of the programs were not able to complete due to scarcity of resources. The results demonstrate that in the expensive cases which might lead to exponential costs, the optimizations are effective in reducing the overheads. In the bad cases in our experiments, all three factors actually grow sub-linearly.

Table 1 records the maximum number of tree nodes, maximum number of program continuations and the maximum size of the data storage (in terms of total number of (variable, value) pairs stored) in each experiment.

	Max Worlds	Max Cont's	Max Data Storage
HO/HP	24	17	6
HO/LP	66	45	6
HO/BP	15	10	7
LO/HP	57	39	51
LO/LP	111	75	63
LO/BP	105	70	22

Table 1. Simulation results

The variation in the numbers in Table 1 has an easy and intuitive explanation due to the different nature of the datasets. The results show that balanced production/consumption gives the smallest tree size, and smallest number of programs. Excessive consumption and low overlapping interests result in larger

tree and more computation since termination becomes less likely. When too many types of resources are produced, the mismatch between production and consumption also becomes more likely. These preliminary results suggest that GCC with optimizations gives reasonable performance under realistic circumstances. They also demonstrate that eventhough GCC is potentially expensive, a "pay only when you use" principle is achievable.

References

1. A. J. Bonner and M. Kifer, "Transaction logic programming", *Intl. Conf. on Logic Programming*, 1993.
2. A. J. Bonner and M. Kifer, "Concurrency and communication in transaction logic", *Joint Intl. Conf. and Symp. on Logic Programming*, 1996.
3. U. Dayal, M. Hsu, and R. Ladin, "Organizing long-running activities with triggers and transactions", *ACM SIGMOD Conf. on Management of Data*, 1990.
4. K. Donnelly and M. Fluet. Transactional events. In *Proceedings of ICFP*, 2006.
5. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.
6. H. Garcia-Molina and K. Salem, "Sagas", *ACM SIGMOD Conf. on Management of Data*, 1987.
7. G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: a survey", *ACM Trans. on Programming Languages and Systems*, 23(4):472–602, 2001.
8. T. Harris, S. Marlow, S. P. Jones, and M. Herlihy, "Composable memory transactions", *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2005.
9. C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.
10. M. E. C. Hull, "Occam - a programming language for multiprocessor systems", *Comput. Lang.*, 12(1):27–37, 1987.
11. J. Jaffar, R. H. Yap, and K. Q. Zhu, "Coordination of many agents", *Intl. Conf. on Logic Programming*, 2005.
12. J. Jaffar, R. H. C. Yap, and K. Q. Zhu. "Indexing for dynamic abstract regions", *Intl. Conf. on Data Engineering*, 2006.
13. R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part I/II", *J, of Information and Computation*, 100:1–77, 1992.
14. G. Nelson, "A generalization of dijkstra's calculus", *ACM Trans. on Programming Languages and Systems*, 11(4):517–561, 1989.
15. R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill, 2002.
16. V. A. Saraswat, *Concurrent Constraint Programming*, MIT Press, 1993.
17. E. Y. Shapiro and A. Takeuchi, "Object oriented programming in concurrent prolog", *New Generation Comput.*, 1(1):25–48, 1983.
18. G. Smolka, "The Oz programming model", *Computer Science Today*, 324–343. 1995.
19. K. Ueda, "Guarded horn clauses", *Intl. Conf. on Logic Programming*, 1986.
20. I. Vlahavas and N. Bassiliades, *Parallel, Object-Oriented, and Active Knowledge-Base Systems*. Kluwer Academic Publisher, 1998.