

0.1 Верификация истории

0.1.1 Кипарис

Для верификации линеаризуемости используется классический алгоритм J. Wing, C.Gong [5] с дополнениями G. Lowe [3] и A. Horn [1]. Приведём краткую схему работы алгоритма.

Определение 0.1 (Модель). Для множеств Ξ, Δ , множества операций $\chi \subset ((\{\perp\} \cup \Xi) \times \Delta)^\Xi$ будем называть $\langle \Xi, \Delta, \chi \rangle$ моделью.

Неформально – модель это набор состояний разделяемого объекта(Ξ) и поддерживаемые операции над ним(χ). \perp означает что операция над данным состоянием объекта не поддерживается. Δ здесь некоторое множество выходных значений соответствующих операциям.

Определение 0.2 (История). Для модели $\langle \Xi, \Delta, \chi \rangle$ будем называть историей последовательность событий вида

- $call.process.op$, где $process$ – идентификатор процесса, $op : \in \chi$ – производимая операция над состоянием разделяемого объекта.
- $ret.process.response$ $process$ – идентификатор процесса, $response \in \{\text{fail}, \text{ok}, \text{info}\} \times \Delta$

Здесь и далее

$fail$ – детерминировано отвергнутая операция,

ok – успешная операция.

$info$ – неизвестно, отвергнута операция или нет(моделирует превышение таймаута и проч.).

Определение 0.3 (Полная история). История является полной, если история, ограниченная на любой процесс, удовлетворяет следующим требованиям:

- История начинается с $call$ -записи.

- За каждой *call*-записью которая не является последней в истории следует соответствующая *ret*-запись, за *ret* записью может следовать только другая *call*-запись.
- Для всех *ret*-записей *status* = **ok**.

Алгоритм оперирует с полной историей, которая может быть получена из реальной удалением детерминировано неудачных операций и **info**-записей и представляет из себя обход некоторого графа состояний.

Также отметим что при проведении экспериментов в случае если операция завершилась с **info** то идентификатор процесса данной операции далее использоваться не может и соответствующий логический процесс необходимо считать завершённым.

Определение 0.4 (Состояние). Будем называть состоянием $\langle state, history \rangle$ где $state \in \Xi$, $history$ – полная история.

Для $\langle state, history \rangle$ будем считать смежными состояния вида $op[state], history \setminus op$ где op – *call* запись принадлежащая $history$ которой не предшествует ни одна *ret* запись, $history \setminus op$ – история с удалённой op и соответствующей *ret*-записью (если таковая есть).

История считается линеаризуемой при достижимости состояния с $history$ без *ret*-записей из начального состояния, в котором $history$ – полная исследуемая история. Корректность данного утверждения доказана в [5].

При постановке экспериментов использовалась реализация из библиотеки `knossos`([2]), запоминающая исследованные состояния в хеш-таблице. Также, перед началом работы алгоритма явно строится граф состояний и переходов для модели. Это уменьшает потребление памяти и избавляет алгоритм от обработки специфичной логики для переопределённой пользователем модели. Также, в хеш-таблице сохраняются не истории а множества линеаризованных операций. Это также ускоряет алгоритм, так как множества представлены последовательностью бит, в силу особенностей современных процессоров такие структуры обрабатываются быстрее чем односвязные списки.

0.1.2 Динамические таблицы

Для верификации snapshot-сериализуемости использовалась модификация вышеупомянутого алгоритма J. Wing, C. Gong, G. Lowe.

При описании алгоритма состояние базы рассматривается как один разделяемый объект со специфичными операциями чтения и записи. Мы объединяем все чтения и старт соответствующие транзакции в одну операцию, а все записи и коммит в другую операцию. В данной постановке необходимо верифицировать отсутствие конфликтующих записей из других транзакций между соответственными чтением и записью.

Далее оперируем с моделью $\Xi = S^n$, для некоторого количества регистров n и множества значений S .

Поддерживаемые операции:

- $start(T)$ Начать транзакцию, прочитайте значения регистров из множества T .
- $commit(T, V)$ Завершить транзакцию, записать в регистры из T значения V .

Верифицируемые истории при ограничении на любой процесс должны являться чередующимися последовательностями $start$ и $commit$ операций. Более того, мы требуем чтобы ограниченные истории начинались с $start$ операции. Чтения вне транзакций можно воспринимать как транзакции, не содержащие записей.

Сформулируем определение snapshot-сериализуемости для описанной модели.

Определение 0.5. Будем называть историю H snapshot-сериализуемой если существует линеаризация [4], в которой для любой пары (u, v) соответствующих $start$ и $commit$ операций, между u и v нет ни одной $commit$ операции которая записывает в регистры, записываемые v .

Первым шагом алгоритм преобразует исходную историю в некоторое множество историй для другой модели $\Xi = (S \times \{0, 1\})^n$.



Рис. 1: Преобразование истории. Каждая транзакция в зависимости от статуса завершения может быть несколькими вариантами преобразована в две операции – чтения и записи соответственно.

Неформально, транзакции будут блокировать регистры в которые планируют писать и соответственно состояние разделяемого объекта теперь включает в себя информацию о заблокированных ячейках. Далее будем говорить о регистрах, которые могут быть заблокированы.

Для каждой пары соответствующих *start* и *commit* операций

- Если *commit* операция завершилась **fail**-записью, обе записи соответствующие *commit* операции удаляются. *start* операция преобразуется в *read* операцию для того-же множества регистров. То есть не меняющее состояние объекта чтение.
- Если *commit* операция завершилась **ok**-записью, то *start*-операция преобразуется в *read* операцию, блокирующую регистры, в которые пишет *commit* операция. Соответственно операция определена только на объектах, в которых соответствующие регистры не заблокированы. *commit* операция преобразуется в *write* операцию, при этом разблокирующую регистры в которые пишет. Соответственно определена только на объектах, в которых соответствующие регистры заблокированы.

- Если *commit* операция не завершилась или завершилась **info**-записью, то рассматриваем два варианта истории, преобразованной одним из вышеупомянутых способов.

Таким образом получаем некоторое множество историй.

Лемма 0.1. *Snapshot-сериализуемость исходной истории равносильна линейизуемости одной из историй, полученных приведённым преобразованием.*

Доказательство. В одну из сторон приведённое утверждение очевидно. То есть из snapshot-сериализуемости очевидна линейизуемость одной из полученных историй. Пусть T получена из H и L – линейизация T . Покажем как построить линейизацию H удовлетворяющую 0.5. Рассмотрим L' соответствующую L . В том смысле, что операции в L' получаются из операций в L игнорированием блокировки регистров. *read*-записям соответствуют *start*-записи, *write* переходят в *commit*. Рассмотрим ограничение L на один из регистров. Также удалим неблокирующие чтения. Из линейизуемости T очевидно, что чтения и записи чередуются, и история начинается с чтения. Покажем, что история устроена следующим образом: каждой записи предшествует соответствующее чтение (с тем же идентификатором процесса), устанавливающее блокировку и за каждым чтением следует соответствующая запись. От противного: пусть пара u, v – последовательные чтение и запись с разными идентификаторами процесса. Тогда в истории должно присутствовать соответствующее v чтение w . Более того, оно должно быть расположено перед u . Тогда после него должна быть запись с другим идентификатором процесса. Получаем противоречие с тем что пара u, v первая. Пусть теперь пара u, v в L' – соответствующие *start* и *commit* операции. Пусть между u и v есть w которая пишет в один из регистров, в которые пишет v . Тогда ограничивая L на этот регистр получаем противоречие вышедоказанному. \square

Алгоритм устроен следующим образом:

- Сначала строится преобразование истории: при неоднозначности преобразования элементом истории является кортеж альтернатив.

У *ret*-записей альтернатив нет. Для некоторых записей помечаем что при выполнении следующую операцию данного процесса надо игнорировать.

- Удаляются все **fail** и **info** записи. Для **fail**-записей необходимо удалить также **call**-запись.
- Производится поиск в пространстве состояний, схожий с описанным в разделе 0.1.1.

Состоянием так же является $\langle state, history \rangle$. При вычислении смежных состояний мы рассматриваем все альтернативы для каждой **call**-записи, и удаляем из истории следующую операцию процесса (если помечено что надо её проигнорировать). При поиске запоминаем пройденные состояния, в том смысле что запоминаем пару (множество удалённых операций, состояние). В [1] запоминали именно пару $\langle state, history \rangle$, что создавало некоторое замедление из-за необходимости сравнения историй, которые хранились в неизменяемом односвязном списке. Авторы предлагают кешировать результаты сравнения списков для ускорения работы алгоритма. Мы же, поддерживая множество удалённых операций (bitset) добились сравнимой производительности при меньшей сложности алгоритма.

Как и в алгоритме, описанном в 0.1.1, при постановке экспериментов, перед началом работы алгоритма явно строится граф состояний и переходов для модели, мемоизируется именно множество удалённых операций, но в отличие от [2], мы используем неизменяемые списки, что с одной стороны замедляет алгоритм, но с другой позволяет эффективно распределить вычисления на несколько процессорных ядер и результирующее время работы (на синтетических тестах) получается меньше.

Номер теста	Длина истории	knossos	наша реализация
1	20	0.2с	2с
2	200	40с	10с
3	800	1200с	100с

Все тесты производились на 16-ядерном процессоре, с доступным объёмом памяти 60G.