Uncomputation

MAISA KIRISAME[*], University of Utah, USA

Program execution need memory. Program may run out of memory for multiple reasons: big dataset, exploding intermediate state, the machine have less ram then others. When this happens, the program either get killed, or the operating system swaps, significantly degrading the performance. We proposing a technique, Uncomputation, that allow the program to continue running even after breaching the memory limit, without a significant performance degrade like the one introduced by swapping. Uncomputation work by turning computed values back into thunk, and upon needing the thunk, computing and storing them back. A naive implementation of uncomputation will face multiple problems. Among them, the most crucial and the most challenging one is breadcrumb. After a value is uncomputed, it's memory can be released but extra memory, breadcrumb, is needed to recompute the value back when needed. The smaller a value is, the breadcrumb it leave will take up a larger portion of memory, until the size of a value is less then or equal to that of the size of breadcrumb, in which uncomputing give no, or even negative memory benefit at all. The more value uncomputed, the more breadcrumb they leave behind, until most of the memory is used to record breadcrumb. This prohibit asymptotic improvement commonly seen in memory-cosntrainted algorithms. We present a runtime system, zombie, implemented as a library, that is absolved of the above breadcrumb problem, seemingly storing recompute information in 0-bits and violating information theory.

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Maisa Kirisame. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Program use memory.

Program may not have enough memory.

Multiple reason: Huge input - opening a file of multiple gigabytes will hang editors

Large intermediate state - model checker, chess bots, etc

Small machine - poor people, embedded device

Even if there is enough memory, still good to use less! Free up memory to run other process/increase cache size/give more room to garbage collectors.

2 PROBLEMS

How to measure size of a Value? (There is sharing)

Partially evict a datastructure - how to reconnect

What value to evict

Breadcrumb

Author's address: Maisa Kirisame, marisa@cs.utah.edu, University of Utah, P.O. Box 1212, Dublin, Ohio, USA, 43017-6221.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

3 API

Zombie provide a monadic api. In particular, we exposed a type constructor, Zombie: * -> *, with the following 3 operations:

return: X -> Zombie X

bindN: Zombie X... -> (X... ~> Zombie Y) -> Zombie Y

get: Zombie X -> X

the type Zombie externally speaking, behave as the Identity Monad - return and get is the identity, and bindN is the reversed apply. however, zombie will uncompute value behind the scene, recomputing them when needed, to save memory.

two things worth noting:

0 - we have a bindN, which take multiple Zombie, and a static function that does not capture any variable, from the inner type to another Zombie. This is different from the usual bindN accepting a single M, alongside a closured argument. This is because a closure will capture non-zombie values, so memory cannot be returned. Imagine the following function: M A -> M B -> M (A * B). An implementation via bind and return will capture A, causing said A to be non-evictable. note that bindN is of the same power as a closure allowing monadic bind.

1 - the get function should only be used to obtain output, that will become a Zombie (or is used in a function that produce Zombie) again. This is because get strip out all metadata zombie maintain, so it is less efficient then obtaining the value via bindN.

4 INSIGHT

There are multiple insights that allow us to solve the above 4 problems. Explaining them aid understanding the implementation of Zombie.

0: treat each cell as independent, ignoring all member relationship. this trivialize size finding.

1: hashconsing. to avoid the same expression producing multiple values, we introduce a type Idx, such that the same trace will be tagged with the same Idx. Zombie X hold Idx instead of X, and the actual values are stored in a datastructure. When return x try to create a Zombie X of tag idx, if idx is stored in the datastructure, that value is used instead, even though the computed value is equivalent to said value. This cause x to be immediately unreachable and can be released from memory. This allow evicting most of the list, only keeping some intermediate value. nowhere reachable via the object graph.

2: note that bind can be nested: the function in bind may call more bind. When such nesting occur, executing the higher-level bind will also re-invoke the lower level bind. Hence - it is not necessary to store the lower-level thunk (even though doing so improve performance, as it will execute less code).

With this in mind, if we can create a data structure imitating the bind/return execution of the program, we can remove non top-level nodes, so that lookup will return a higher-level result then the remove node, we solved the breadcrumb problem.

5 TOCK TREE API

Below is an API summing up what we need.

Tock = u64

Range = Tock * Tock

Make: TockTree X

Insert: TockTree X -> Range * X -> ()

Per two call to Insert, their range must either be nested or non-overlapping.

Lookup: TockTree X -> Tock -> Option (Range * X)

Remove: TockTree X -> Range -> ()

6 IMPLEMENTATION

Combining the 3 insights above we now sketch an implementation of Zombie. There is a global tock, which value begin from 0, increasing on every invocation of bind/return. A Zombie<X> hold an Integer, tock, instead of value of type X. The value is stored in the tock tree. Whenever a call to return or bind is finished, we put a Node onto the tock tree. The range of said node is the tock value at the begin and end of the invocation.

For return, the node contain the value.

For bind, the node contain the list of input Zombie, the output Zombie, and the function pointer.

In bind, we have to force values from Zombie X to X. This is done via looking it up from the tock tree. If the look up node is not a Value, but the Thunk, we need to rewind the tock to the beginning of said thunk, replay the value, get the value from the tock tree (todo: avoid infinite loop here), and restore the tock.

6.1 replaying semantic

7 PROOF

8 CACHE POLICY

9 EVAL

10 OLD

All Zombies are threaded through a logical clock, a tock, stored as a u64. During execution, the clock increases whenever makeZombie and bindZombie are called.

A Zombie contains, theoretically speaking, contains only the tock that the Zombie is created at. To access the actual value of the Zombie, a ZombieNode, we rely on a global data structure. It will be explained later. (We also store a weakpointer to ZombieNode as a cache, to quicken access. This is just a cache, and is irrelevant to the rest of Zombie). A ZombieNode holds X but is inherited from the Object class, which contain a virtual destructor and nothing else. This essentially allow one to type erase ZombieNode of different type, to put them into the same data structure.

We implement Zombie via two global data structures, a log and a pool. The pool track all currently in-memory Zombie representation - it manage space The log record actions taken used for recomputation - it manage time

The pool holds a vector of uniqueptr<Phantom>, the class that manage eviction. It has api to evict objects, and api to score the profitability of eviction. When we are low on memory, we can find a value in this vector, call evict(), and remove it. This will free up memory in the log.

For every call to makeZombie and bindZombie, we keep track of when the function is called vs when the function exited. This establishes a tock range. For makeZombie, since it does not call any more functions, its interval length is 1, containing exactly the tock of the zombie. Tock range overlap iff one contain another, iff one function calls another function. We then store tock range, paired with information on said function. For makeZombie we store a shared pointer to the ZombieNode. For bindZombie, we record the function used to compute it, as std::function<void(const std::vector<const void*>)>, alongside the tocks of all input Zombies, and the tock of output Zombie. We call it a Rematerializer. The input argument is type erased to void*, as Zombie is type polymorphic, but we need a type uniform interface here.

The log is some kind of interval tree. Each node in the tree stores its begin and end interval, alongside a value, and a BST from tock to sub nodes. The key of BST is the start of the sub node's interval. When we query the log via a tock, we will find a value with the closest interval containing said tock. This mean, we might query for a Zombie, but instead of getting a ZombieNode, we get a Rematerializer. In such a case, we replay the Rematerializer by fetching the ZombieNode from the tock (replaying recursively if necessary), setting the clock to begin of the function temporarily, and entering said function. Afterward we set the clock back, and fetch the value from a tree. When we set the clock via rematerialization, we always set it to a smaller value, thus guaranteeing termination.

The log itself, when replaying, also serve as a memo table, indexed by tock. When we want to create a Zombie, we will skip when such Zombie already exist in the log. When we want to run BindZombie, we return the result tock when the precise Rematerializer exist in the log. This allow us to not compute the same function multiple time, if we know its result, and allow us to not store the same Zombie multiple time.

Note that all non-top-levels node in the tree can be removed. When we do so, we will free up memory, and when we need the value again, we can always replay a node higher up.

This design allow us to store n log n amount of metadata per n alive(nonevicted) objects.

Thus, when a Rematerializer dominate no value, and another value dominate Rematerializer, we remove it. (todo: implement). Idea: manage this using pool.

There is a slight error with the above design: maybe the value is recomputed and put on the tree, but then evicted, and fetch will not return a ZombieNode but a Rematerializer. To fix this we introduce a global data structure called Tardis, consist of a tock and a shared ptr<ZombieNode>*. When we create Zombie, when the tock match that of tardis, it will write to the shared ptr, holding the value longer.

Recursive object: Zombie allow recursive object, e.g. a list, a tree, or a graph. For those object, a part of the object can be evicted, and sharing does not affect size counting. This feature is implemented by ignoring recursiveness. All nesting need to loop through Zombie, and when it does that, the management of subsequent nested Zombie is completely handoff to it. Note that this allow keeping some value of the list in memory, even when there is previous node evicted!

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009