# 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 memory then others, etc. When this happens, the program either get killed, or the operating system swaps, significantly degrading the performance. We propose a technique, uncomputation, that allow the program to continue running smoothly even after breaching the memory limit, without a significant performance degradation. Uncomputation work by turning computed values back into thunk, and upon re-requesting 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 that of 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, so uncomputing save no memory. 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, 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), 7 pages. https://doi.org/XXXXXXX.XXXXXXX

### 0.1 Motivation(1.5pg)

## 1 INTRO

Program consume memory. Program might use too much memory:

- The input might be huge. Editing a file require loading the whole file into memory; Autocomplete load and parse the whole codebase; Image editors will keep multiple replica of similar images.
- Exploding intermediate state: Model checker and Chess bot search through a huge state space, Compiler and Interprocedural analysis expand the AST, Deep Learning/Backpropagation create and keep intermediate value.
- Low memory limit. wasm have a memory limit of 2GB, embedded device or poor people may have device with lower ram. A personal computer might be running multiple programs, which must contend for a fixed memory limit.

Even when there is enough memory, it is still beneficial to use less memory. Operating System use memory for file cache, the program may have software cache, garbage collector fire less often give more free space, and extra space could be used to increase load or to execute other process.

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

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

Why not buy more memory?

However, the only generic technique for reducing memory consumption, swapping, come with great drawback. Swapping move a chunk of data from main memory onto disk, and upon re-requesting it, move the data from disk back into main memory. It degrade performance significantly as disk is far slower then CPU and main memory, and it is especially bad for functional programs as most values is functional programs are boxed, they typically have less cache locality, exacerbated by swapping which operate on pages granularity.

We propose uncomputation, a technique to reduce memory consumption for arbitrary purely functional programs. When needed, uncomputation select some values in the object graph according to cache policy, reverting it back into thunk, releasing memory until it is needed again, in which case it will recompute said values and replacing the thunk with the values.

The idea of uncomputation is not new, ... But there is no fully automatic generic solution.

We introduce a library, zombie, which enable uncomputation for arbitrary purely functional program.

### 1.1 Guarantee(0.5pg)

There are multiple guarantees our approach give.

Correctness guarantee: uncomputing will not effect the final result (preservation), and will not be in an infinite loop (progress)

Asymptotic guarantee: O(1) access time for all in-memory objects, and O(1) space for each in-memory objects (O(0) space for uncomputed objects)

### 1.2 Key Idea(1pg)

I dont like this section. feels like spoiler. There are 3 key insights that enable the above guarantee:

- Use a monadic API this is a bit week but is a dependency. A monadic API give a generic yet turing-complete better word method of extending a language. Furthermore it immediately ask an interesting question: what is the meaning of Zombie<Zombie<X»? likewise, what is join?
- A global clock increasing on return/bind invocation. This not only give a way to quickly detect same objects for hashconsing, but as opposed to classical hash consing, the key itself have semantic meaning: < denote happens-before.
- Recomputing A also recompute what is inside A. This mean we can forget about some inner thunk, as long as we can redirect the pointers to such thunk to an outer thunk. One possibility is to track backpointers and fix them, but this take significantly more time and space. Instead, Track function exit time too. Entering and exiting form a range, and two separate range either form a nested relationship or disjoint relationship. We can then design a datastructure, such that lookup failure return a match one-level above.

## 2 UNCOMPUTATION(5PG)

### 2.1 Motivation

### 2.2 Strawman

#### *2.2.1 Semantic.*

### 2.3 Guarantee

### 2.4 Problems

While the high-level idea seems straightforward, there are multiple subtle questions:

- Recursiveness. A value, for example, a list, might be recursive. Furthermore, there might be sharing inside such datastructure. In such a case, how do we measure the size of a value, which is a valuable statistics guiding what to uncompute?
- Partialness. Suppose a list is generated via anamorphism. We might want to uncompute the head of the list, but keeping some intermediate node inside the list. When recomputing said list, we need to be able to retrieve such intermediate node, to avoid spending extra time to recompute them, and extra memory to store them twice.
- Uncomputation candidates. Which value should we pick to uncompute?
- Breadcrumb. After a value is uncomputed, we need to store information needed to recompute said value. This become a bottleneck when most value is uncomputed, or when each value is small.

We present Zombie, a library for uncomptuation, alongside solutions to the questions above.

#### *2.4.1 recursiveness.*

#### *2.4.2 doppelganger.*

#### *2.4.3 breadcrumb.*

## 3 PLANT VS ZOMBIE(8PG)

### 3.1 Tock Tree

### 3.2 Recomputation

### 3.3 Correctness Gurantee

#### *3.3.1 Progress.*

#### *3.3.2 Preservation.*

### 3.4 Asymptotic

### 3.5 Cache Policy(2pg)

## 4 EVAL(3PG)

## 5 CASE STUDY(2PG)

## 6 RELATED WORK(2PG)

### 6.1 Memory-Constrained Algorithm

Memory Limit is a common problem. Lots of different subfields of CS had experienced such problem, and developed specialized algorithm that trade space for time, by recomputing, for their own need. cite. In particular, a famous algorithm, TREEVERSE, had been re-discovered independently multiple times in unrelated subfields.

Keeping uncomputation at each subfield not only cause multiple re-discovery and re-implementation, wasting valuable researcher/programmer time, but also come short in integration. Suppose a complex, memory-hungry software have 2 sub-parts, both of which are memory hungry. How much uncompute responsibility should each part take? What happens if two part depend on each other?

### 6.2 Pebbling

Space time tradeoff is a known-topic in Theoretical Computer Science. It is mostly modeled as a "pebble game" quickly explain. It had been known that this problem is NP-complete and inapproximable. There are also generic result in this space. Hopcroft prove that for any program that take N time, it could be modified to take O(N / log(N)) space. However, the proof is non-constructive, and thus does not help building an automatic runtime like ours.

Our work mostly focus on the overhead problem - how to minimize the overhead of recording metadata needed for recomputation. How to make an API that is generic yet embeddable. In another words - our work is orthgonal to advances in pebble game. In fact, one could imagine replacing our cache policy with some of those algorithms.

We also provide a greedy cache policy that is both fast and make reasonably good decision on the list of benchmark, both real and synthetic, provided below, since the theoretical result is weak and non-constructive. There are some specialized result of pebble game on planar graph and on tree. One could imagine upon detecting planar graph/tree, switching from our cache policy to the theoretically optimal ones.

### 6.3 Memoization

Uncomputation is the dual to Memoization.

### 6.4 Garbage Collection

GC collect provably dead value, but we can also collect not-provably dead value, and even alive value.

It is like optimistic lock vs pessmistic lock.

### 6.5 Compression

Uncomputation is, fundamentally speaking, a form of compression.

Pigeon hole principle mean it is fundamentally impossible to have a generic compression algorithm. Uncomputation exploit the inductive bias that a program pointer is smaller then its memory consumption is larger.

## 7 MAYBE?

### 7.1 Constant Optimization

### 7.2 Tail Call

### 7.3 Non Tail Call

## 8 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.

## 9 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.

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!

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.

## 10 TOCK TREE API

we need a data structure, such that when after removing a node, looking said node up does not return Nothing, but rather return a less precise answer.

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 -> ()

geometric series

## 11 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.

### 11.1 replaying semantic

Program evaluate differently in replaying mode then in non-replay mode, as relevant Node might already be on the tock tree. When

Every entering tock (lhs of a range) uniquely identify a return/bind.

$$A - Formula \qquad Longer - Formula \qquad And \qquad The - Last - One \qquad \frac{\begin{array}{c} aa \\ bb \end{array}}{dd \qquad ee \qquad ff}$$

$$\frac{\dfrac{aa \qquad bb}{cc} \qquad dd}{ee}$$

## 12 PROOF

### 12.1 Formal Definition

### 12.2 no infinite loop

### 12.3 replaying give old value back

### 12.4 it is same as a normal interpreter

## 13 CACHE POLICY

note that the cache policy does not effect the correctness of the implementation, only the performance.

GDSF

### 13.1 Measuring compute time

### 13.2 recursive cache policy

### 13.3 union find

## 14 EVAL

## 15 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.

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