Uncomputation

MAISA KIRISAME*, University of Utah, USA
PAVEL PANCHEKHA*, 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 than 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 gracefully even after breaching the memory limit, without 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 some memory, breadcrumb, is needed, so we can recompute the value back. Ironically, in a applicative language, due to boxing all values are small. This mean uncomputation, implemented naively, will only consume more memory, defeating the purpose. 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 and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 6 pages. https://doi.org/XXXXXXX.XXXXXXX

bullet points:

(1) The importance of saving memory
(2) Pros of recomputation
(3) The perlis of breadcrumbs (no idea how to make this positive)
(4) Our approach leave no breadcrumb. formally state Guarantee
(5) taba: explain
(6) evicting a context
(7) multiple eviction and no-breadcrumb
(8) picture of eviction
(9) replaying the context
(10) recursive replaying
(11) picture of replaying
(12) Ordering all value reference (pointer) by logical time (tock)
(13) Replay objects (checkpoints): thunk, input, storage
(14) Approximate data structure: return largest key <= k
(15) (Recursive) Replaying
(16) Base Language (purely functional lambda calculus)

Authors' addresses: Maisa Kirisame, marisa@cs.utah.edu, University of Utah, P.O. Box 1212, Salt Lake City, Utah, USA, 43017-6221; Pavel Panchekha, , 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

(17) abstract machine
(18) The approximate data structure (splay list)
(19) Bit counting
(20) The abstract Machine
(21) Double O(1)
(22) Correctness property
(23) Time and Space measurement
(24) Union Find
(25) GDSF
(26) Unrolling
(27) Benchmark explanation
(28) Setup
(29) Result table
(30) Explain Result
(31) Related Work(2pg)
(32) Conclusion

adapton bullet points:

(1) abstract
(2) other work: IC(traces)
(3) cant do sharing/swapping/switching/interactive application
(4) key concept: demand computation graph
(5) formalization sneak peak
(6) some numbers
(7) spreadsheet example
(8) show ref/get/set/thunk/force and demand computation graph
(9) graph about sharing
(10) subconcept: dirtying the dcg
(11) why dirtying do swapping
(12) textual description of the calculus (CBPV)
(13) the changes to CBPV
(14) greeks definition
(15) describing trace
(16) greeks type
(17) greeks operational semantic
(18) text operational semantic
(19) text type
(20) patching a trace
(21) formal syntax of spreadsheet example
(22) API, and explaining memo
(23) the core change propagation algorithm
(24) explaining change propagation
(25) why change propagation better then other approach
(26) explaining setup for lazy/swapping/switching/batch
(27) baseline and what is the machine setup
(28) table of result
(29) talk about some benchmark
(30) explaining result

Classical PL runtime	Zombie
Address	Tock
Value	Value
Address Space	Tock Tree
Constructing Values	Constructing Values and saving Thunks
Reading from Address	Querying the Tock Tree
Garbage Collection	Eviction
Reading freed address: Impossible/Segfault	Recomputation

Table 1. Side by side comparison between classical PL runtime and Zombie

(31) overhead
(32) spreadsheet example - explain what is a spreadsheet
(33) what it do to spreadsheet
(34) talk about number'
(35) IC
(36) self adjusting computation
(37) FRP
(38) conclusion

1 INTRO

2 OVERVIEW

3 CORE LANGUAGE

Zombie works on a untyped, purely functional, call by value language. Program in the language then executed by the cek abstract machine.

Below we sketch out the language and the cek machine. Note that this is the standard semantic - there is neither uncomputation nor replaying present. Uncomputation will be represented independently afterward.

```
Name = N = A set of distinct names
  Expr = E := N | Left E | Right E | Case E N E N E | Prod E E | Zro E | Fst E
  | Lam N E | Let N E E | App E E
  The source language
```

```
Continuation = K := Stop | KLeft K | KRight K | KCase Env N E N E K | KProd0 Env E K | KProd1
V K | KZro K | KFst K | KLet N Env E K | KApp0 Env E K | KApp1 V K
  Environment = Env = (N, V)...
  Value = V := VLeft V | VRight V | VProd V V | Clos Env N E
  State = Step Expr Env K | Apply K V
  step: (Expr, Env, K) -> (Expr, Env, K)
  step(N, Env, K) goto apply(K, Env(N))
  step(Left X, Env, K) goto step(X, Env, KLeft K)
  step(Right X, Env, K) goto step(X, Env, KRight K)
  step(Case X LN L RN R, Env, K) goto step(X, Env, KCase LN L RN R Env)
  step(Prod L R, Env, K) goto step(L, Env, KProd0 K R)
  step(Zro X, Env, K) goto step(X, Env, KZro K)
```

```
step(Fst X, Env, K) goto step(X, Env, KFst K)
step(Let A = B in C, Env, K) goto step(B, Env, KLet A K C Env)
step(App f x, Env, K) goto step(f, KApp0 K x)
step(Lam N E, Env, K) goto apply(K, Clos Env(fv)... N E)
apply(KLeft K, V) goto apply(K, VLeft V)
apply(KRight K, V) goto apply(K, VRight V)
apply(KCase Env LN L RN R K, VLeft V) goto (L, Env(LN := V), K)
apply(KCase Env LN L RN R K, VRight V) goto (R, Env(RN := V), K)
apply(KProd0 Env R K, V) goto (R, Env, KProd1 V Env K)
apply(KProd1 L K, V) goto apply(VProd L V, K)
apply(KZro K, VProd X Y) goto apply(X, K)
apply(KFst K, VProd X Y) goto apply(Y, K)
apply(KLet A Env C K, B) goto (C, Env(A := B), K)
apply(KApp0 Env X K, V) goto (X, Env, KApp1 V K)
apply(KApp1 (Clos Env N E) K, V) goto (E, Env(N := V), K)
Abstract Machine
```

4 UNCOMPUTING AND RECOMPUTING

4.1 Tock

The critical insight of zombie is that multiple abstract machine state compute the same value. To be more precise, if a machine state x step to a machine state y, x must have computed all value that y might compute, and possibly more. This indicate that we do not have to store all previous machine state - some might be dropped in favor of older ones.

For this purpose, we introudced a global, logical time of 64 bit int, a tock. Tock start at 0, and increase by 1 on each transition(step/apply) in the abstract machine, and whenever a value is constructed. Conversely, all tock smaller then the current tock correspond to either a value, or a executed machine state, and vice versa.

More importantly, given a value that correspond to tock X, any machine state with tock Y < X will recompute it. The largest Y under the constraint will do the least amount of transition computing said value.

4.2 Tock Tree

To pair a value with it's tock concretely(I mean in the runtime, the word look bad), and to allow a value to be recomputed, we abstract over the memory space(need better words), replacing pointers to value, to tocks instead. The actual values are stored on a global data structure, the tock tree. Reading from a pointer is replaced from querying the tock tree with the tock. The tock tree additionally store machine state as they are executed, so a value might be uncomputed and recomputed in the future with any earlier machine state.

The tock tree is a binary search tree with the crucial property that lookup returns the largest node with key <= the input. This allow us to drop any node in the tock tree, with the exception of the leftmost node. Each node on the tock tree correspond to an execution of a transition, and contain:

(1) The starting tock of the transition execution, t
(2) An array of cell(actual value), created during the transition, of corresponding tock t+1, t+2...
(3) The state it transit to.

Note that it store the transit-to state, but not the transit-from state, for that state is useless. Additionally, two tock is stored: the begin tock, t, and the end-tock, implicitly stored in the State. It is technically possible to compute the end-tock by adding t + 1 with length of the array.

Uncomputing is then merely deleting a non-leftmost value from the tock tree.

Formally speaking,

$$before : Value = V := VProdVV|... \qquad after : Value = V = Tock \qquad Cell = ProdVV|...$$

$$before : \qquad State = StepExprEnvK|ApplyKV \qquad after :$$

$$State = StepExprEnvKT|ApplyKVT$$

4.3 Replay

During execution, the tock needed to be converted back to a Cell. It proceed as follow:

(1) to convert tock t to a Cell:
(2) query the tock tree on t to get a Node
(3) if the cell is in the array, return said Cell
(4) otherwise, issue a replay to t.

A replay t suspend the current machine state, replacing it with the State in the Node returned from tock tree, and executing until the tock reach t. The old state is then resumed with the Cell at t. Replay form a stack: a replay might more replay.

$$Replay = R := NoReplay|ReplayingTR$$

5 IMPLEMENTATION

5.1 Tock Tree

To exploit the temporal/spatial locality, and the 20-80 law of data access (cite?), the tock tree is implemented as a slight modification of a splay tree.

This design grant frequently-accessed data faster access time. Crucially, consecutive insertion take amortized constant time.

The tock tree is then modified such that each node contain an additional parent and child pointer. The pointers form a list, which maintain an sorted representation of the tock tree. On a query, the tock tree do a binary search to find the innermost node, then follow the parent pointer if that node is greater then the key. This process is not recursive: the parent pointer is guaranteed to have a smaller node then the input key, as binary search will yield either the exact value, or the largest value less then the input, or the smallest value greater then the input.

5.2 Picking Uncomputation Candidate

Note that the guarantee we prove is independent of our policy that decide which value to uncompute (eviction policy).

5.2.1 Union Find.

5.2.2 The Policy.

5.2.3 GDSF.

5.3 Language Implementation

For implementation simplicity and interoperability with other programs, zombie is implemented as a C++ library, and the Cells are ref-counted. Our evaluation compiles the program from the applicative programming language formalized above(give name), to C++ code.

5.4 Optimization

5.4.1 Fast access path. Querying the tock tree for every value is slow, as it requires multiple pointer traversal. To combat this, each Value is a Tock paired with a weak reference, serving as a cache, to the Cell. When reading the value, if the weak reference is ok, the value is return immediately. Otherwise the default path is executed, and the weak reference is updated to point to the new Result.

5.4.2 Loop Unrolling. To avoid frequent creation of node object, and their insertion to the tock tree, multiple state transition is packed into one.

5.5 Bit counting

6 FORMAL GUARANTEE

6.1 Safety

Evaluating under replay semantic give same result as under normal semantic

6.2 Liveness

Evaluating will eventually produce a value
Decreasing on lexicalgraphic ordering on the replay stack do work

6.3 Performance

memory consumption is linear to amount of object with O(1) access cost

7 EVALUATION

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