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), 2 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 IMPLEMENTATION

5 PROOF

6 EVAL

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