重新发明 Self Adjusting Computation

这是我阅读 Self Adjusting Computation 的 thesis [1] 的前几章后，尝试从要解决的问题出发，推导出 thesis 里的算法的记录。

1 Overview

我们有某个问题的一个静态算法。现在，我们希望把这个算法自动变成 dynamic/kinetic 的。首先，对于初始输入，无疑静态算法已经是最优的了。所以，我们要做的是，在静态算法第一次运行后，收集并记录它上一次运行中得到的各种中间结果，并利用这些中间结果来加速输入变动后的下一次运行。

我们有经典的 CEK/CESK 抽象机。我们知道，在程序的一次运行中的每一个中间状态，都可以用下面的内容来（唯一地）确定：

- 数据流（Environment/Store）
- 控制流（Control + Kontinuation，或者说 PC + call stack）

对于一个程序的某一次具体的 program trace，数据流图是完全静态的。因此，在静态算法运行一次后，我们可以给上一次运行的所有 dataflow edge 赋予一个全局的标号（location），这样一来，所有数据流就都能打包进一个 store 里。因此，程序的某次 program trace 的每个中间状态都可以表示成一个 (C, S, K)（Control=PC + Store + Kontinuation=call stack）的三元组。

现在，我们已经得到了“静态算法的上一次运行”的一种数据化的表示：程序的一次运行，就是在一串 (C, S, K) 的 checkpoint 之间移动的过程。接下来，我们需要考虑如何利用这些 checkpoint 来加速算法的下一次运行。

2 推导 Dynamic Dependence Graph（DDG）

首先，如果严格按照小步语义，程序的一次运行中会有超级多的 checkpoint。这其实没有必要，checkpoint 只是起到一个缓存的作用，保存太多 checkpoint 反而浪费性能。因此，我们可以删掉很多 checkpoint，只保留一些“关键”的 checkpoint，把上一次运行切分成更粗粒度的 block。Self Adjusting Computation 中的切分方式是：

[1] https://www.cs.cmu.edu/~rwh/students/acar.pdf

- 保证每个 block（两个 checkpoint 之间）不要跨越函数。即在每次函数调用的开始和结束都保留一个 checkpoint。这没什么好解释的，允许跨函数的 block 太鬼畜了
- 在每一团 data flow in-edge（连续的 `read` 操作）处插入一个 checkpoint。这样每个 block 的 data flow 依赖都可以 trivial 地得到

这个切分方式看上去就很合理，就不做太多论证了。

确定要保留哪些 checkpoint、如何切分程序后，就需要考虑如何利用这些 checkpoint 来搭建缓存。下面列一个很自然但行不通的思路：

- 直接记录上一次运行中有哪些 (C, S, K) 的 checkpoint，下一次运行时，只要运行时遇到相同的 checkpoint，说明程序运行的最终结果没变，可以直接结束运行。
  - 这么做的坏处显而易见，cache 命中率堪忧。只要程序的最终结果变了，cache 就不可能命中，约等于屁用没有

所以，我们不应该用完整的 (C, S, K) triple 做缓存的 key：这样命中率太低了。这里，我们可以引入第一个重要优化：对每段程序的数据流依赖做分析。如果整个程序的输入变化了，但是某个具体的 block 依赖的输入没变化，那么这个 block 的运行结果，也就是下一个 checkpoint 的 (C, K)，也不会变化。这样一来，我们就可以跳过这个 block 内部的代码的运行。按照这个思路，可以得到第一个版本的算法：

- 对于程序上一次运行的每个 block，我们以它的起点的控制流 (C, K) 为 key，记录：
  1. 这个 block 的数据流依赖，以及它们在上一次运行中的取值
  2. 上一次运行中，这个 block 结束时的 (C, K)
- 第二次运行时，每进入一个 checkpoint (C, S, K)，我们检查缓存中是否存在一个 (C, K) 的 entry。如果存在，而且 C 依赖的数据没有变化，就跳过 C 的运行，并进入缓存中记录的下一个状态（由于所有 dataflow edge 的取值被打包到了一个巨大的 store 里，对于一个 immutable 的程序，如果某个 block 的输出没有变化，什么都不做即可）

上面这个算法已经能节省不少工作量了。前后两次运行中，相同的 block 内部的计算可以被省掉。但它还不够好。首先，每次缓存命中，我们只能前进一格。其次，每进入一个新的 checkpoint，都需要检查一下缓存。为了解决第一个问题，观察到，如果连续的 N 个 block 的数据流依赖都没有变化，那么当我们进入第一个 block 开头的 checkpoint 时，完全可以安全地跳到第 N 个 block 的结尾。由此，可以得到一个改良版的算法：

- 我们记录上一次运行中的所有 checkpoint 的 (C, K)，以及每个 block 的数据流依赖。我们搭建一个从 location/data flow edge 到依赖它的 block 的反向缓存

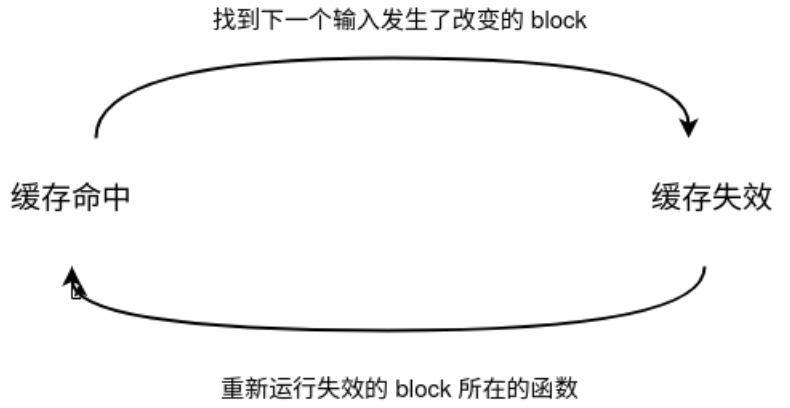
- 在第二次运行时，我们记录哪些 data flow edge 发生了改变。当我们进入一个 checkpoint (C, K) 时，如果它在上一次运行中也出现过，那我们直接找到上一次运行中下一个输入发生了变化的状态，并直接跳转到它

这样一来，我们可以直接跳过一大片连续的、输入没有发生改变的 block。缓存的利用效率就提升了。但这个改良版的算法还是有一个问题，就是它每次进入一个 checkpoint 都需要去检查缓存。上面的算法中，缓存的 key 是控制流 (C, K)。当程序的数据输入发生改变，它接下来的运行的控制流也可能发生改变，和上一次运行失去同步。我们不知道什么时候控制流才会恢复同步，就只能在每个 checkpoint 都看一下缓存有没有命中。

但上述情况不是必然的。事实上，在没有 control effect 的程序中，有许多天然的 control flow 同步点：函数调用。只要程序进入了某个函数，它就一定会从这个函数返回。进入函数的点和从函数返回的点，它们的 call stack 一定是一样的。所以，在上面的算法中，假设由于输入发生改变，我们必须重新运行某个 block，我们知道，哪怕输入发生了变化，程序也一定会从这个 block 所在的函数返回。那时，前后两次运行的控制流一定会恢复同步，缓存也一定会再次命中。所以，我们可以给上面的算法加这样一条补充：

- 当输入发生了改变，我们需要重新运行某个 block 时，直接重新运行这个 block 所在的函数调用。重新运行后，缓存一定会命中

这样得到的算法，正是 Self Adjusting Computation thesis 中描述的 Dynamic Dependence Graph（DDG）。这个算法可以一次跳过一大片缓存命中的 block，而且重新运行输入改变的 block 时，不需要在每个 checkpoint 都检查缓存是否命中。

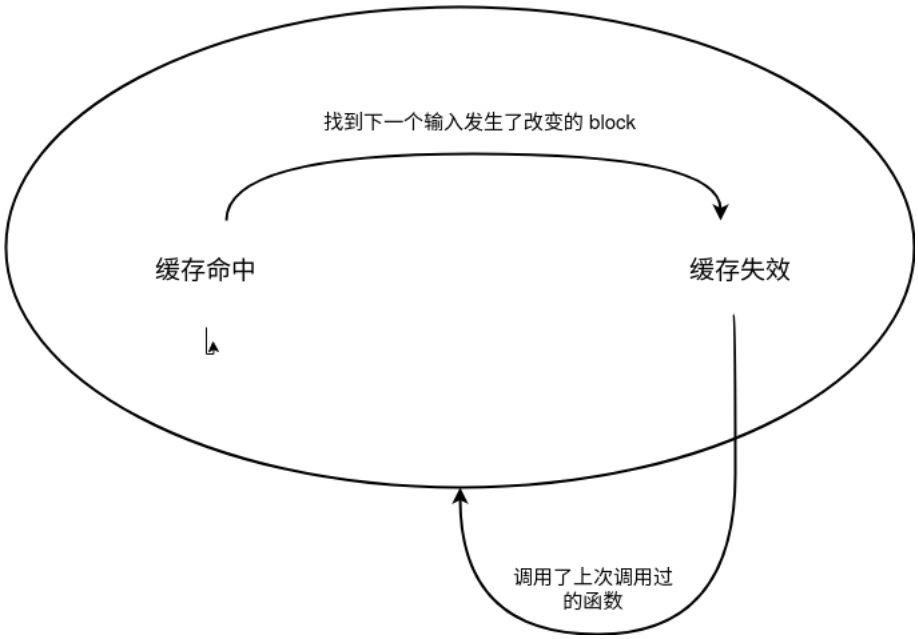


3 推导 Memorized Dynamic Dependence Graph（MDDG）

DDG 虽然好，也有一个问题：缓存的 invalidation 范围太大了。假设上一次运行最外层的函数调用的输入变了，那么相当于缓存全部失效，整个程序都要重

新跑。因此，我们希望能在程序中的某些节点依然主动地检查缓存，来提高缓存的利用率。最简单的做法是，沿途记录程序的 call stack K，并在想要检查缓存的 checkpoint 把当前 block 的代码 C 和 K 一起拿去查缓存。但这样做有一些实现上的问题。代码 C 只需要全局标个号就是一个标量。Dataflow edge 扔到一个全局 store 里之后就变成了 memory location，也是一个标量。但 call stack 是一个栈,它的长度不是常数的,所以拿 call stack 去做 key 查缓存感觉有些不太好。

这里，我们可以再次利用上函数调用的良好性质。一次完整的函数调用在 call stack 上是 relocatable 的：无论输入是否改变、进入函数时 call stack 如何，离开函数时 call stack 一定没变化。所以，如果在第二次运行时程序调用了一个函数 f，而第一次运行时也调用了同一个函数 f，我们可以把对 f 的调用抽出来，当成一段完整的程序，然后利用上一次运行中调用 f 的信息，来加速 f 的运行：



这样一来，每次检查缓存就只需要用当前函数（C）做 key 了，不再需要用 call stack 做 key。这样得到的算法，正是 thesis 中的 Memorized Dynamic Dependence Graph（MDDG）。

4 剩下的细节

如果你读了原 thesis，或许会觉得上面的描述和原 thesis 的算法还差了不少东西。比如原 thesis 的 virtual clock 去哪了？但其实按照上面的思路去推导如何实现上述算法，很容易就能推导出原 thesis 中剩下的构造。这里以 virtual block 为例：

- 上述算法中，我们需要知道各个 checkpoint 之间的相对顺序关系，这样才能找到“第一个输入发生了改变的 block”来重新运行。重新运行一个 block 可能导致后续的 block 的输入也发生改变, 所以按正确的顺序重新运行是很重要的。为此，我们可以给每个 checkpoint 一个全局的 timestamp
- 我们希望缓存命中的 block 的所有记录都不需要改变，包括 timestamp。然而，同一段程序在不同的输入下控制流可能不同，产生的 timestamp 数量也不同。所以，如果想要在重新运行程序中间的某一部分后还保持 timestamp 的正确顺序，朴素的整数 timestamp 是不够用的
- 上面就是原 thesis 中 virtual block 的由来。按照需求去推一下就能得到和原 thesis 一模一样的 virtual block 结构