最终改进性能的部分，我主要是加特判来做。

举例：

在block.c文件的new\_item函数中，最初我首先做这个事情：

// maybe increase runtime: find all existing item and to see whether suitable for updating

// free item\_id and space enough

for (idx; idx < block->n\_items; idx++) {

item\_id = get\_item\_id(block, idx);

avail = get\_item\_id\_availability(item\_id);

size = get\_item\_id\_size(item\_id);

if (avail && size >= item\_size) {

block->tail\_ptr -= item\_size;

offset = get\_item\_id\_offset(item\_id);

for (i = 0; i < item\_size; i++) {

block->data[offset - 3 \* sizeof(short) + i] = item[i];

}

get\_item\_id(block, idx) = compose\_item\_id(0, block->tail\_ptr, item\_size);

return idx;

}

}

就是在一个block中，给他添加新item时，遍历已有item判断是否有item符合可更新要求：available且size大于新入item的size。如果有则更新返回。

在此之后我再判断block可用空间是否足够，做如今代码内的事情。

后面我把这段删掉后速度快了一些（也可能和测试环境相关）。我new\_item时只考虑block已有位置够不够，不去管是否有item可被更新。

在hash\_map文件中，我给很多种情况都事先加了特判，当特定情况触发时（比如说insert找到某个table中存在目标addr，记录后直接return，避免许多不必要的反复搜索。

这种特判看似简单，其实加多了之后对整体性能还是优化很多的。Hash\_map test，最终经过优化代码大概快了60多秒。测试比较稳定。

更新策略选用LRU算法，也应当是理论上最快的。

以下是最终两次测试大致性能图：



