FCMalloc: A Fast Memory Allocator for Fully Homomorphic Encryption

Akira Umayabara (Advisor: Prof. Hayato YAMANA)

Computer Science and Engineering, Graduate School of Fundamental Science and Engineering, Waseda University, Japan

uma@yama.info.waseda.ac.jp

category: graduate

Abstract

In multi-core systems, a scalable memory allocator is indispensable to guarantee fast execution of many threads running simultaneously. There exist general-purpose scalable memory allocators such as TCMalloc, JEmalloc, and SuperMalloc, however if we know the memory allocation usage in advance, we can speed up more. In this paper, we propose FCMalloc which enables to control local heaps among threads by handling memory pools at physical memory level with pseudo free technique. Experimental evaluation with frequent pattern mining algorithm, Apriori, for fully homomorphic encryption shows 2.4 times faster execution speed in comparison with JEmalloc which is the fastest one in the previous allocators.

Categories and Subject Descriptors D.4.2 [Operating Systems]: Storage Management—Allocation/deallocation strategies;

Keywords Memory Allocator, malloc, free, Virtual Memory, Physical Memory

1 Introduction

In multi-core systems, calling dynamic memory allocation functions, such as malloc(3) and free(3), require long time, especially when many threads simultaneously allocate memory. To speed up such memory allocations, JEmalloc [1], TCMalloc [2], and SuperMalloc [3] have been proposed. Though these memory allocators are proposed for general purpose to release as much memory as possible for providing memory to other running applications, we can have more speedup by controlling memory pools effectively when only one application whose memory usage is statically known runs.

Especially, applications of fully homomorphic encryption (FHE) [4] satisfy the above condition. In the big data era, secure computing becomes important to outsource both data storage and calculation to the cloud. In such situation, FHE is only the solution to realize the secure computation. FHE is a public key cryptography where we are able to perform arbitrary computation over ciphertexts without decryption. When we adopt FHE, no comparison is possible during the calculation, because we are not able to adopt pruning technique, which usually decrease the required total amount of memory. That is, it helps us to know the total amount of required memory in advance,

In this paper, we propose FCMalloc which reduces the locking cost during the memory allocation by controlling fully connected local heaps without extra system calls for memory management.

This work was supported by JST CREST Grant Number JPMJCR1503, Japan.

2 Related Work and Proposed Method

The comparison of general-purpose memory allocators and our proposed method is shown in Table 1 and Figure 1. As shown in Table 1, glibc malloc is not scalable because of its legacy implementation. Though TCMalloc before 2.2rc does not release virtual or physical memory pools, TCMalloc 2.2rc or after, JEmalloc and SuperMalloc release only physical memory pools. This is because virtual memory space is not as expensive as physical memory space in large memory space machines such as 64bit machines.

Our idea is to adopt pseudo free technique not to release physical memory pools besides virtual memory pools by controlling local heaps among threads so that we are able to handle allocation and deallocation of memory in user space, which is called FCMalloc. Especially, when an application requires repetition of allocation and deallocation of memory, we can have large speedup. More precisely, pseudo free technique and controlling local heaps among threads enable FCMalloc to reuse the committed virtual memory, which was already attached to physical memory, without memory remapping by OS. As shown in Figure 1 (f), though the number of communications among local heaps increases, decreasing the locking cost will have advantage. Moreover, we are able to decrease the number of context switching because of enlarging the execution time in user space, without extra system calls for memory management.

In our proposed method, a certain amount of memory is allocated in advance followed by dividing it into a set of local heaps each of which is assigned to a single thread so that each thread can access without locking. A local heap is divided into main-heap and sub-heap. Both heaps have size-segregated singly linked lists of the memory head pointers. To manage the linked lists, FCMalloc rounds up the requested size of memory to the nearest power of two. The main-heap keeps only the memory allocated at the current running thread, while the sub-heap keeps the memory allocated at other threads. The sub-heap keeps it for a while followed by transferring it, with locking, to another main-heap where the memory was originally allocated. In this way, the number of size-segregated singly linked lists in each local heap is balanced.

In FCMalloc, malloc(3) returns the head pointer of memory popped from an appropriate linked list of the running thread's local main-heap, and free(3) pushes memory to an appropriate linked list of the running thread's local main-heap or sub-heap depending on where the memory was originally allocated.

3 Experiments and Discussion

In the experiments, we chose a FHE application because we know its memory usage in advance. By using HElib [5], we implement Apriori algorithm [6] whose flowchart is shown in Figure 2. It consists of three parts: initial processing, main processing, and terminal processing. In the initial processing, it creates a ciphertext

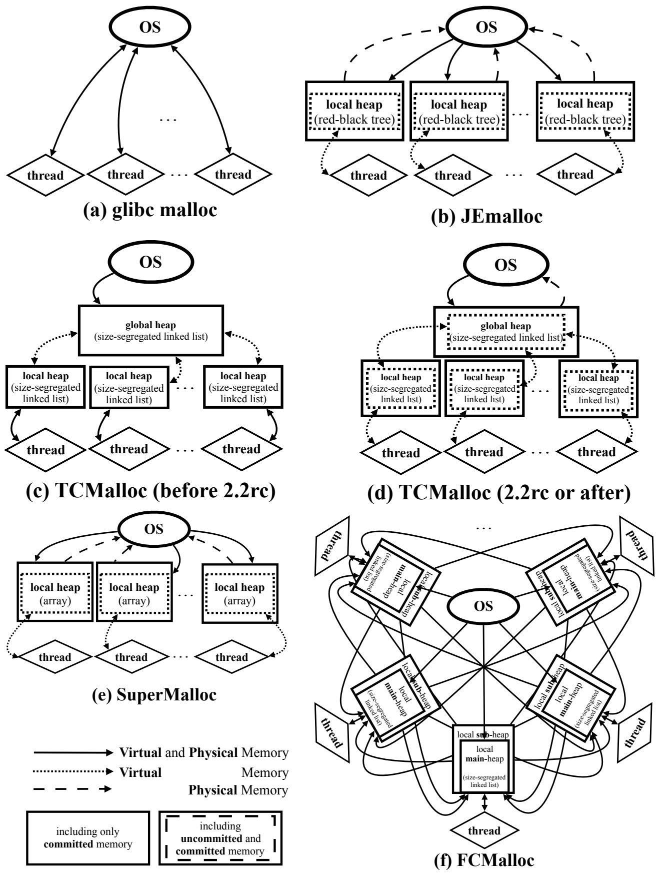
database. The main processing consists of FHE multiplications that are divided into the two steps: Ctxt modDownToLevel method and Ctxt tensorProduct method. Here, Ctxt is a class of C++, which represents a ciphertext in HElib. The terminal processing destroys the ciphertext database.
The experimental environment is shown in Table 2. Table 3 shows the parameters. The result is shown in Table 4 that excludes SuperMalloc because it requires calling madvise with MADV_HUGEPAGE equipped with Linux 2.6.38, which was not implemented in our platform.
In the experiment, we compared the main processing time, because it becomes large when we set small minimum support value which is an important parameter of Apriori. As shown in Table 4, we have confirmed that FCMalloc successfully speeds up 2.4 times in comparison with JEmalloc which is the fastest one in the previous allocators. The time of freeing ciphertexts of TCMalloc or JEmalloc is so-called 3s pause problem [3]. FCMalloc solves this problem by pseudo free technique. Instead, FCMalloc has a disadvantage of large memory usage.

4 Conclusions

We propose FCMalloc to control local heaps to speed up memory allocation and deallocation by adopting pseudo free technique. By applying FCMalloc to a FHE application, we achieved 2.4 times speedup in comparison with JEmalloc. Our future work includes a reduction of memory usage.

References

[1] Google Inc., “gperftools: Fast, multi-threaded malloc() and nifty performance analysis tools”, http://code.google.com/p/gperftools/ accessed on 2017-03-15.
[2] Evans, J., “A Scalable Concurrent malloc(3) Implementation for FreeBSD”, Proc. of BSDCan, 2006.
[3] Kuszmaul, B. C., “Supermalloc: A Super Fast Multithreaded Malloc for 64-bit Machines”, Proc. of ISMM, pp. 41-55, 2015.
[4] Gentry, C., “A Fully Homomorphic Encryption Scheme”, Ph.D. Thesis, Stanford University, 2009.
[5] HElib, http://shaih.github.io/HElib/ accessed on 2017-03-15.
[6] Imabayashi, H., et al., “Secure Frequent Pattern Mining by Fully Homomorphic Encryption with Ciphertext Packing”, Proc. of the 11th Int’l Workshop on Data Privacy Management (DPM), LNCS vol. 9963, pp. 181-195, 2016.



OS
thread
(a) glibc malloc
local heap (red-black tree)
(b) JEmalloc
global heap (size-segregated linked list)
local heap (size-segregated linked list)
(c) TCMalloc (before 2.2rc)
(d) TCMalloc (2.2rc or after)
local heap (array)
(e) SuperMalloc
local sub-heap
local main-heap (size-segregated linked list)
(f) FCMalloc
Virtual and Physical Memory
Virtual Memory
Physical Memory
including only committed memory
including uncommitted and committed memory

Figure 1. Comparison of Memory Allocators

Table 1. Comparison of Memory Allocators

Name (Version)	Virtual memory is freed.	Physical memory is freed.	Global heap exists.	Local heaps exist.	Communication between local heaps exists.
glibc malloc	Yes	Yes	No	No	-
JEmalloc[1]	No	Yes	No	Yes	No
TCMalloc[2](before 2.2rc)	No	No	Yes	Yes	No
TCMalloc[2](2.2rc or up)	No	Yes	Yes	Yes	No
SuperMalloc[3]	No	Yes	No	Yes	No
FCMalloc(Proposed)	No	No	No	Yes	Yes

Table 2. Experimental Evaluation Environment

Name	Value	Name	Value
CPU model	Intel(R) CPU E7-8880 v3 @ 2.30GHz	OS	CentOS 6.7 (64bit)
# of cores	72	Linux version	2.6.32-504.30.3
Memory size	1TB	g++ version	4.9.2
L1, L2, L3 cache	32KB, 256KB, 45MB	g++ opt. option	-O2

Table 3. Parameters used in Experimental Evaluation

Parameter	Value	Description
p^r	11^7	plain text space
k	80	security parameter (default value)
l	8	# of levels in the modulus chain
c	3	# of columns in the key-switching matrices (default value)
w	64	hamming weight of the secret key (default value)
N_{slot}	915	# of slots
N_{item}	18	# of items
N_{trans}	1,943,424	# of transactions

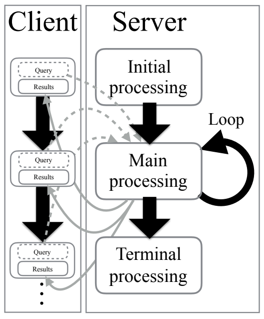


Figure 2. Flowchart of Client and Server Processing of Apriori

Table 4. Experimental Result

	Name	glibc malloc	JEmalloc [1]	TCMalloc [2]		FCMalloc (proposed)
	Version	2.12-1	3.6.0	2.0	2.5	-
	Initial processing time [sec.]	301.6	59.7	88.6	107.0	83.7
Main Processing	Time of modDownToLevel method [sec.]	6.6	6.0	17.3	35.0	9.6
	Time of tensorProduct method [sec.]	350.0	22.4	94.6	118.9	5.8
	Time of freeing ciphertexts [sec.]	0.25	9.3	79.2	4.2	0.08
	Total [sec.]	356.9	37.7	158.1	191.1	**15.48**
	Terminal processing time [sec.]	0.9	0.5	0.4	6.3	0.02
	Maximum memory usage [GB]	197.4	199.9	208.5	208.8	306.1