Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 1953510, 12 pages
https://doi.org/10.1155/2022/1953510

Research Article

Study of Energy-Efficient Optimization Techniques for High-Level Homogeneous Resource Management

Suman Mann,[1] Nitish Pathak,[2] Neelam Sharma,[3] Raju Kumar,[4] Rabins Porwal ,[5] Sheelesh Kr Sharma,[6] and Saw Mon Yee Aung [7]

[1]*Department of Information Technology, Maharaja Surajmal Institute of Technology (MSIT), GGSIPU, New Delhi, India*
[2]*Department of Information Technology, Bhagwan Parshuram Institute of Technology (BPIT), GGSIPU, New Delhi 110078, India*
[3]*Department of Computer Science and Engineering, Maharaja Agrasen Institute of Technology (MAIT), GGSIPU, New Delhi, India*
[4]*Department of MCA, G. L. Bajaj Institute of Technology & Management, Greater Noida, India*
[5]*Lal Bahadur Shastri Institute of Management, Delhi, India*
[6]*Department of MCA, GNIOT Engineering College, Greater Noida, India*
[7]*Department of IT, Technological University (Toungoo), Toungoo, Bago Region, Myanmar*

Correspondence should be addressed to Rabins Porwal; rabins.porwal@lbsim.ac.in
and Saw Mon Yee Aung; drsawmonyee.aung@gmail.com

Received 29 April 2022; Revised 4 July 2022; Accepted 12 July 2022; Published 27 July 2022

Academic Editor: Ajay Rakkesh R

Copyright © 2022 Suman Mann et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Resource management efficiency can be a beneficial step toward optimizing power consumption in software-hardware integrated systems. Languages such as C, C++, and Fortran have been extremely popular for dealing with optimization, memory management, and other resource management. We investigate novel algorithmic architectures capable of optimizing resource requirements and increasing energy efficiency. The experimental results obtained with C++ can be extended to other programming languages as well. We emphasize the inherent drawbacks of memory management operators. These operators are intended to be extremely generic in their application, just as the concept of dynamic memory is. As a result, they are unable to take advantage of the various optimization techniques and opportunities that specific use cases present. Each source code file is modeled after its own distinct memory usage pattern, which can be used to speed up memory management routines. Such concepts are frequently time-consuming and costly to implement; consequently, they are not the primary concern of application developers, as they require manual development and integration. We intend to address this gap by providing a suite of memory management algorithms that enable dramatic performance improvements at the source code level while allowing for seamless integration across multiple use cases. The techniques have been evaluated on several performance parameters, and results have been presented. In this paper, we have compared a variety of memory allocation techniques and compared their space and energy efficiency requirements. Three variants of SSDAM, SSDAM-E, and DLLOM strategies have been evaluated and compared against the base performance of new and delete operators. SSDAM-E, SSDAM with new delete operators, and DDLOM improve the memory consumption by the factors of 8.01, 7.0, and 4.0, respectively. In the worst case, SSDAM-E gave an average running time of 5.650 sec faster than the DLLOM average time of 7.496 sec. As far as energy efficiency is considered, SSDAM-Original and SSDAM-E-Original attain 100%, in comparison with the base efficiency of 12.48% characterized by new/delete operators.

1. Introduction

For many software applications, dynamic memory management (DMM) has become prohibitively expensive. According to studies, C-programs can consume up to 30% of the system's operating time in memory release and allocation. Object-oriented programming (OOP) frequently results in additional work and removal. According to the data, C++ programs share memory more than similar C programs. The causes, however, are unknown. At this time, no data

sharing patterns for C++ applications have been reported. This emphasizes the importance of quantitative analysis in allocation patterns in order to achieve the best possible system structure. This paper introduces a novel approach to investigating memory allocation at the source code level. We begin by classifying all of the conditions that may necessitate the use of dynamic memory management (DMM) in C ++. These memory allocation patterns, according to our theories, are linked to the C++ system or language. DMM requests from builders, copywriters, or overloaded operator overload, for example, are not the same as OOPs in C++. Members of a program requesting a new or deleting operator have functions that are directly related to the program. A novice C++ programmer, on the other hand, can easily write a C++ program without using the object-oriented paradigm.

New allocation strategies focusing on parallel allocation [1], the spread of multilayered architecture, and the use of multithreaded applications have been proposed in recent decades. Following the introduction of 64-bit programs and the widespread acceptance of large-memory applications, the fragmentation problem, which had previously received little attention in allocator design, has emerged as a major issue that will degrade both space efficiency and performance.

Current memory allocators, in particular, are focused on using fast memory allocation and deallocation, and they all use the same process to organize virtual memory in multiple bins. The hoard memory allocator [2] portion, for example, has 32, 64, 128, 256, and 512 byte bins. We will give it 64 bytes from a 64-bit bin if we want to allocate 47 bytes. This method of memory allocation is clearly faster, but there are 64-47 = 17 bytes wasted.

With apps that allocate less memory, this design has worked well in the past. However, if we have a resource-intensive application that allocates in the same way, the waste is enormous. This massive waste not only improves space efficiency, but it also causes virtual memory explosion, resulting in more TLB misses [3], which will severely impede performance. In response to this new problem, this paper proposes a new heap or memory allocator design that focuses on fragmentation reduction. We concentrate on large memory allocation and provide them with the exact size they want to split in order to reduce TLB loss [3, 4] and improve performance. Experiments show that, when compared to Hoard, our new memory allocation design can earn up to 1.3x performance (average over 28.8%) with less memory usage (18%) on large memory-footprint benchmarks that share multiple items, indicating great potential for widespread acceptance. Our memory allocation is a common memory allocation that can be used in a variety of applications, including logic control programs and scientific computing.

2. Literature Survey

DMM has proven to be a cost-effective component in the majority of programming languages. Memory allocation and deallocation define the overall efficiency of many software systems, as described by Michael Neely [5]. He and Zorn [6] demonstrated that memory-intensive programs consume up to 40 percent of the runtime to allocate and free memory. Memory allocator has a significant impact on program efficiency in terms of performance and memory space [7]. According to related research, managing dynamic objects is just as simple as allocating and dealing with them.

Maas demonstrated a novel approach to memory fragmentation and object lifetime management during program execution. On several production servers, he reduced memory fragmentation by up to 78 percent by only using huge pages. Allamanis [8] demonstrated that as the number of objects grows, so does execution time and memory fragmentation. He graphed the results and found a logarithmic curve that showed a direct relationship between objects and execution time. Existing C/C++ memory allocators use a number of strategies to reduce average fragmentation in C/C++ programs [9]. Several methods for solving this problem are evaluated, and only some of them are found to be useful [10]. These methods turned out to be inherently limited and inapplicable in all situations. Robson demonstrated that allocators can suffer from large memory fragmentation during various experiments, which can have negative consequences for the program's overall efficiency and even result in program crash or failure [11]. Cohen and Petrank use partial compaction to prove upper and lower bounds on defragmentation [12, 13].

To reduce fragmentation, TCMalloc, a well-tuned allocator [14], was used to organize and compare execution time versus object size.

Its current heap profiling mechanism does a good job of identifying long-lived objects by generating a list of sampled objects at the end of the application's execution, the majority of which are long-lived, including their allocation sites. Installing an HTTP handler accessible by paper of [15], an open-source profiling and analysis tool was used to compare the results. This made it possible to compare how many of these allocations were allocated and deallocated on the same CPU or thread. The result is saved into a protocol buffer at the end of a sampling period. Bayesian [16] simulated various scenarios and came up with a result that predicted object lifetime during program execution using various strategies and optimization techniques. Languages like LISP and Java have had garbage collection for a long time [17–19]. Compaction is implemented as part of the trash collection algorithms in modern runtimes such as the Hotspot JVM, the.NET VM, and the SpiderMonkey JavaScript VM [20]. The fact that no single GC provides the simplest results for all programs motivates these efforts. In terms of approach, this line of work gives the developer no control and prevents the mixing of different GC designs within the same program. Shoaib et al. [21] described a concept called the Write Rationing GC in Big Data Processing, which moves objects with a large/small number of writes into DRAM/NVM to extend the lifetime of the NVM. NVM for

managed programs is supported by approaches like Espresso [22]. Nowadays, C++ programs, as opposed to C programs, make extensive use of dynamic memory for short-term allocations, which often results in faster access to objects and thus increases program efficiency [23]. In comparison to C programs, studies have shown that C++ programs invoke dynamically created objects much faster and with fewer errors [24].

The challenges that modern memory management systems face are exemplified by Memcached. In modern web architectures, Memcached is widely used for caching temporary data [25]. Facebook and Twitter, for example, make extensive use of the technology to reduce database server load and rely on a 99 percent hit rate to scale to their massive user bases [26].

Automatic cached memory cleanup in mobile apps, as described by Umar Farooq [27], can reduce program complexity to a much greater extent and aid in the smooth running of apps. As a result, using multiple CPUs to increase a system's computation power is usually ineffective and becomes a bottleneck [28]. C and C++ programs rely heavily on dynamic memory allocation, and the related key functions (malloc() and free() for C, new() and delete() for C++ programs) have long been required parts of standard libraries. The researchers are now working on creating a standard dynamic memory allocation technique/algorithm that is more efficient in terms of speed, performance, and memory than previous ones. There exist many applications of efficient resource management including testing of object-oriented software [29], multimedia optimizations [30, 31], and mathematical optimizations [32]. Memory and energy efficiency become a prominent criterion in many scenarios, such as system on chip (SoC) [33], edge computing, federated machine learning, cluster computing, and Internet of everything I(OE). Sundari et al. [33] discuss energy-efficient SoC memory management techniques. Some recent energy-efficient static and dynamic memory management techniques can be found in the works given in [34–36].

3. Design Goals

Performance: the first objective is to create a memory manager that outperforms the memory managers included with the default language. Concurrent memory allocations and deallocations must not cause any performance degradation.

Novelty: the memory manager should be upgraded to manage repeated assignment patterns in the code and to optimize their performance accordingly.

Platform independent: the memory manager design should be independent of any particular system and should be portable across platforms without relying on platform-specific dependencies.

Ease of use: when users incorporate a memory manager into their code, they should only need to change a small amount of code.

Robustness: the memory manager must not leave any traces after its use has ended and must restore the requested traces before the system terminates. This prevents the memory from being leaked. The memory manager should handle all error cases.

4. Strategies Used in Design

Request memory in large chunks: during program startup and then intermittently during code creation, one of the most popular memory management techniques is to request large memory combinations. Memory allocation requests for specific data structures are documented in these frameworks. As a result, fewer system calls are made and operating time is increased.

Allocation pattern optimization: in any system, certain request sizes for specific applications are more prevalent than others. Your memory manager will perform admirably if it is optimized to better handle these requests.

Memory deallocation to optimize operating system calls: during execution, memory should be integrated into containers. Additional memory requests should be serviced by these containers. If the call is unsuccessful, memory access should be transferred to one of the large chunks allocated during the startup process. While memory management is intended to improve system performance and prevent memory leaks, this approach may result in a smaller program's memory footprint due to the reuse of deleted memory.

5. Implementations and Performance Analysis

The default new and delete operators in C++ for allocating and deallocating memory have some limitations, which we can overcome by writing an optimized and efficient memory management algorithm/code that makes use of the concepts of computing, caching, memory, and data structures that we have learned thus far. Operator new executes in a nondeterministic manner. When we call new, the operating system may or may not allocate a new physical page to the process, which can be quite slow if we do so frequently. When new is called, system looks for a memory block large enough to hold our request. Additionally, we discuss memory fragmentation.

For example, if we allocate 10 KB from the middle of a 20 MB chunk, we cannot allocate the remaining 20 MB in one go. If we access a memory region but do not free it, we have a memory leak. If there are infinite memory allocation operations, the system's memory will be rapidly depleted, and the system will crash. New and delete operators consume a significant amount of time for allocation and deallocation purposes, which can have a greater impact on the speed of a C++ application at a higher level.

For instance, suppose we are given the task of creating 1000 objects and are required to create and destroy them 500000 times in a given context. This equates to $500000 \times 1000 \times 2$ (allocations + deallocations). If we use the default new and delete operators for this purpose, the benchmarking of the preceding example results in a processing time of 30.469 seconds on a particular computer.

```
1  class MyPracticalClass
2  {
3  public:
4  int a, b;
5  void initialize(int a, int b)
6  {
7  MyPracticalClass::a = a
8  MyPracticalClass::b = b
9  }
10  }
11  int main()
12  {
13  —- start clock time
14  MemoryManager<MyPracticalClass, 1000 > memMan
15  MyPracticalClass *arra[1000]
16  for (int i = 0; i < 500000; i + +)
17  for (int j = 0; j < 1000; j + +)
18  {
19  arra[j] = memMan.nxtAddress()
20  arra[j]- > initialize(i, j)
21  }
22  for (int j = 0; j < 1000; j + +)
23  memMan.freeAddress(arra[j]);
24  —- end clock time}}
```

Algorithm 1: General pattern for all allocators that are discussed.

We discovered that for this particular problem of allocating and disposing of approximately 1000 objects in a cycle, we can effectively reuse more than 70 percent of the objects.

This can be accomplished by reusing memory and employing compact, contagious data structures. Our approach would be to create a memory manager object using templates to determine the type of class for which we will create objects and also to pass it the number of objects to create in a single cycle (in our example 1000). In our example, we will use a simple user-defined class that the memory manager will allocate and deallocate.

Now we will discuss Algorithm 1, which is the benchmarking routine that the following memory managers will follow.

(1) Memory Manager::nxtAddress(), returns the address of memory. The memory size is equal to the class size (sizeof operator), here MyPracticalClass

(2) MemoryManager::freeAddress(), is supposed to destruct the memory pointed by the pointer and later reuse that memory for another object allocation

(3) For getting the running time of all the implementations we will be using chrono time library of C++

(4) All allocators request memory from operating system using either malloc or new operator. In our implementations we are using malloc

5.1. *SSDAM: Single Size-Determined Array Memory*

5.1.1. Idea. The main idea behind implementing this technique was to request a large chunk of contagious memory and use some portion of this memory to serve an object whenever required

Figure 1: Two objects allocated in the memory pool.

Figure 2: Memory pool full of objects.

Figure 3: Memory pool becomes free of objects after each cycle.

Also, we will reuse the memory whenever the previously allocated object does not require it anymore.

5.1.2. How It Works. We will first initialize a large chunk of memory. The size of this pool is equal to the size of class (here MyPracticalClass) multiplied the number of objects (here 1000) we will be allocating and deallocating in one cycle (here no. of cycle is 500000). Whenever we call MemoryManager::nxtAddress() function, it returns a pointer to a memory address of sizeof(MyPracticalClass) anywhere from the pool. We can think of the pool as an array of empty objects and size of this array we have already calculated above (here 1000). The empty objects serve as the memory for actual object we want to create and use. Whenever we want to create an object, we call the function MemoryManager::nxtAddress().

```
1  MemoryManager(class T, count) {
2  typeSize = sizeof(T)
3  /* malloc returns address of memory */
4  ref = malloc(count * typeSize)
5  /* type cast address of memory (pointer) to type T*/
6  startRef = <T* > ref
7  /* initialize nxtRef as address of object of type T just before the address of startRef. It is analogous to -1index in arrays */
8  nxtRef = startRef–1
9  /* endRef is analogous to index equal to array length + 1 in arrays */
10  endRef = startRef + count}
11  nxtAddress()
12  {
13  ++nxtRef
14  /* if the pool is used up, reset nxtRef to point to first object memory in the pool */
15  if (nxtRef == endRef)
16  nxtRef = startRef
17  return nxtRef
18  }
19  freeAddress(objPtr) {
20  objPtr- > destructor()
21  }
22  MemoryManager() {
23  free(ref)}
```

Algorithm 2: SSDAM.

Figure 4: Single complex memory node.

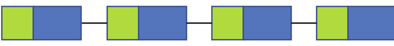

Figure 5: Multiple complex memory nodes in doubly linked list chain.

This returns the memory for the object. We then initialize the object using initialize member function on the class. The memory manager now sees that the memory has been served to some object.

Next time whenever we want to allocate memory for another object the memory manager returns the next memory address that is free. Figure 1 shows the allocation of two objects in the memory pool.

In our example, we have 1000 allocations in one cycle so after all the memory address are returned, our pool will look like as shown in Figure 2.

Now, the next we do is 1000 deallocations. This is done by calling MemoryManager::freeAddress() with memory address of object as parameter, i.e., pointer to the object. After 1000 deallocations, the SSDAM pool will look like this.

Now, 1 cycle out of 500000 cycles of 1000 allocations and deallocations is done. In the next cycle, this memory manager's pool can be reused to do 1000 allocations and deallocations. As one can see, SSDAM follows the principle of reusing memory and using compact and contagious data structure. Memory pool becomes free of objects after each cycle as shown in Figure 3.

5.1.3. Benchmark. We benchmarked the standard new/delete approach using the same machine and environment conditions as discussed previously. The conventional new/delete method resulted in an average running time of 30.469 sec-

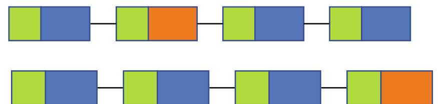

Figure 6: Moving free node to the end of chain.

Figure 7: Multiple free nodes.

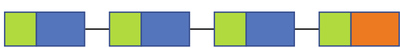

Figure 8: Doubly linked list after using the first free node.

Figure 9: Doubly linked list with no free nodes.

Figure 10: Adding new complex node at the end of the chain.

Figure 11: All nodes in the chain are free.

```
1  MemoryManager(class T, poolObjCount, count)
2  {
3  typeSize = sizeof(T)
4  /*prev points to complex node before to current complex node*/
5  /* next points to complex node next to current complex node*/
7  TYPE Link { prev, next}
8  linkSize = sizeofPoolLink
9  /* size of complex node */
10  typePlusLinkSize = typeSize + linkSize
11  sRef = <link * > malloc(typePlusLinkSize)
12  sRef- > prev = sRef- > next = null
13  freeLink = eRef = sRef
14  }
15  nxtAddress() {
16  objMemoryPointer = null
17  if (freeLink) {
18  objMemoryPointer = <T* > freeLink + 1
19  freeLink = freeLink- > next}
20  /* create new complex node */
21  else
22  {
23  tmpLink = <Link * > malloc(typePlusLinkSize)
24  tmpLink- > prev = eRef
25  tmpLink- > next = null
26  eRef- > next = tmpLink
27  eRef = tmpLink
28  objMemoryPointer = <T* > tmpLink + 1
29  }
30  return objMemoryPointer
31  }
32  freeAddress(objPtr) {
33  objPtr- > destructor()
34  tmpLink2 = (<Link * > objPtr)-1
35  tmpLink = tmpLink2- > prev
36  tmpLink3 = tmpLink2- > next
37  /* next complex nodes exist */
38  if (tmpLink3) {
39  /* previous complex nodes also exist */
40  if (tmpLink) {
41  tmpLink- > next = tmpLink3
42  tmpLink3- > prev = tmpLink
43  }
44  /* previous complex nodes does not exist so currently tmpLink2 must be the first complex node inchain*/
45  else {
46  sRef = tmpLink3
47  tmpLink3- > prev = null
48  }
49  /* move tmpLink2 to end of chain as free complex node */
50  eRef- > next = tmpLink2
51  tmpLink2- > prev = eRef
52  eRef = tmpLink2
53  }
54  tmpLink2- > next = null
55  if (freeLink == null)
56  freeLink = tmpLink2
57  }
58  MemoryManager() {
59  tmpLink = sRef
60  while (tmpLink){
61  sRef = sRef- > next
```

Algorithm 3: Continued.

```
62 free(tmpLink)
63 tmpLink = sRef
64 }
65 }
```

Algorithm 3: DLLOM.

onds. The SSDAM technique resulted in an average running time of 3.800 seconds. As a result, after a few tweaks to the general way C++ code is written, the program appears to be eight times faster. This can have a noticeable effect on performance. Response times will be shortened, resulting in improved service.

5.1.4. Pseudocode

5.2. DLLOM: Doubly Linked List Optimized Memory

5.2.1. Idea. The idea behind this implementation was to use linked lists instead of arrays and see the changes during benchmarking. This means that, from our principle of reusing memory and using compact and contagious data structures, we will not be using compact and contagious data structures here. Instead, we will be using linked list memory. We will have a number of nodes equal to the number of objects that the user wants to create, and these are linked using a doubly linked list. Each node has two pointers for referring to the back and front nodes. Along with these two pointers, there is a data field. This data field is what we will be using to store the information for a memory address where we can initialize our object.

5.2.2. How It Works. As in the case of SSDAM, we have initialized a memory pool at the beginning before executing the main code. But in DLLOM, we are not initializing or creating a node. Instead, whenever we want to create an object, we call the MemoryManager:: nxtAddress() function, which creates a node, saves its address in a doubly linked list chain, and returns the value of the created node's data field. The data field is the pointer to the memory size of the object we want to create.

The doubly linked list is now represented as follows.

This node has a blue part also, not only green because the green portion fully describes the doubly linked list. Hence, DLLOM is not purely a doubly linked list, but rather a complex doubly linked list node. So, the complex node memory returned can be divided into two parts. Use the first part to store doubly linked list data, and the second part is reserved for our object space. Now, the memory manager will be maintaining the chain of these complex nodes. Use the doubly linked list to link the complex nodes and the other parts of the complex node as a memory pool for our single-class object. Further objects are allocated in a similar fashion, and the complex chain can now be represented as shown in Figures 4 and 5.

In the case of deallocation of objects, whenever an object is freed at any part of the chain, it is pushed to the end of the chain. This complex node can now be reused in the sense that its object memory pool (orange part) can be used to serve memory requests for another object allocation. When there is another object deallocated, the list becomes as shown in Figure 6.

Multiple free nodes can be seen from Figure 7. Now, whenever we request the memory manager to return memory, it will look for the first free link at the end of the chain and return the object memory. Then, the list will look like the one shown in Figure 8.

After next allocation, no free complex nodes are left (Figure 9).

Since no free complex nodes are left, this means that if there is one more memory request to memory manager then it will have to create a new complex node and then return the memory from that complex node. A new complex node is added at the end of the chain (Figure 10).

Figure 12: Single complex node in the singly linked list chain.

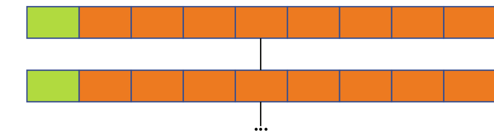

Figure 13: Multiple complex nodes in the singly linked list chain.

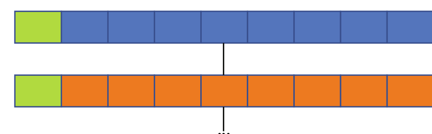

Figure 14: Using up the first complex node in the chain for allocations.

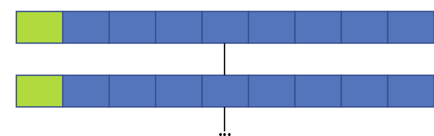

Figure 15: Singly linked list with no free complex nodes.

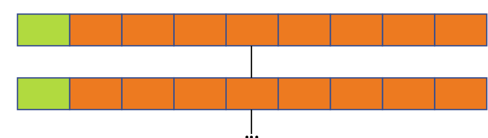

Figure 16: Singly linked list with all free complex nodes.

```
1  MemoryManager(class T, poolObjCount, count) {
2  typeSize = sizeof(T) 3/*4 ref points to actual pool of objects memory (of size poolObjCount * typeSize) in complex node.5 next
points to complex node next to current node.6 */
7  TYPE PoolLink ref, next
8  poolLinkSize = sizeof(PoolLink) 9/*
10   sRef, eRef and cRef refer to start, end and current complex nodes, respectively
11   Size of one complex node is poolObjCount * typeSize + poolLinkSize.12 */
13   sRef = eRef = cRef = <PoolLink * > malloc(poolObjCount * typeSize + poolLinkSize)
14   sRef- > ref = sRef + 1
15   sRef- > next = null
16   noOfPools = count/poolObjCount
17   /* we have already created one pool above. So create total pools -1 */
18   poolsToCreate = noOfPools -1
19   PoolLink *p
20   while (poolsToCreate) {
21   p = <PoolLink * > malloc(poolObjCount * typeSize + poolLinkSize)
22   p- > ref = p + 1
23   p- > next = null
24   eRef- > next = p
25   eRef = p 26}27/* point to first object memory in the memory pool of current complex node */
29   startRef = <T* > cRef- > ref
30   nxtRef = startRef -1
31   endRef = startRef + poolObjCount 32}
33   nxtAddress() {
34   ++nxtRef 35/* if current pool is full jump to next pool */
36   if (nxtRef == endRef) {
37   cRef = cRef- > next 38/* all pools are full. Jump to first pool */
39   if (cRef == null)
40   cRef = sRef
41   startRef = <T* > cRef- > ref
42   nxtRef = startRef
43   endRef = startRef + poolObjCount 44}
45   return nxtRef 46}
47   freeAddress(objPtr) {
48   objPtr- > destructor() 49}
50   MemoryManager() {
51   PoolLink *tmpLink = sRef
52   while (tmpLink) {
53   sRef = sRef- > next
54   free(tmpLink)
55   tmpLink = sRef
56   }
57   }
```

Algorithm 4: SSDAM-E.

This feature of DLLOM where it can allocate as many nodes as it requires makes the DLLOM chain of some specific length and hence does not restrict the number of objects that can be allocated in a given cycle. So DLLOM is more flexible than SSDAM in serving any number of memory requests. For deallocation of 1000 objects, the process of freeing and pushing the freed complex node to the end of the chain is repeated 1000 times. After that, as shown in Figure 11, we have all the free nodes in the chain.

Now, 1 cycle out of 500000 cycles of 1000 allocations and deallocations is done. In the next cycle, this memory manager's pool can be reused to do 1000 allocations and deallocations in the way we have discussed above.

5.2.3. Benchmark. The DLLOM approach gave the average running time of 7.496 sec. So, it is slower than SSDAM but it is still faster than general new/delete approach by a factor of around 4.

5.2.4. Pseudocode

5.3. SSDAM (Single Size-Determined Array Memory–Extended)

5.3.1. Idea. The main idea behind this implementation was that the SSDAM memory manager was only initializing one pool of memory, say of a size equal to the number of objects in one cycle multiplied by the size of a single object (in our

Table 1: Performance of the implementations in contrast to traditional allocators.

Implementation	Variation	Best case time	Worst case time
Default (new/delete)	Original	1x	—
SSDAM	Original	8.01x	—
SSDAM	Placement new	7.4x	—
SSDAM	New/delete operator overloading	7.0x	5.3x
SSDAM-E	Original	8.01x	—
DLLOM	Original	4.0x	—

case, 1000 ∗ sizeof (MyPracticalClass)). If we wanted to say more than 1000 objects in one cycle, say 1000000, the operating system will either return memory or not. In the event that it does not return memory, a runtime error will be thrown, or if it returns memory, the memory space that we think is contagious might not be physically contagious, which can degrade our program performance in terms of more CPU request cycles, more indirections, and probable cache misses. Now to deal with that SSDAM, instead of requesting a single large contagious pool of memory, request multiple small contagious pools of memory and connect those using a singly linked list.

5.3.2. How It Works. Small contagious pools of memory connected using singly linked list will be managed by memory manager and use it to serve memory request for our object allocations. Similar to the concept of complex node in DLLOM approach, we also have complex nodes in SSDAM-E. Shown in Figure 12 is a single complex node of SSDAM-E single linked list.

So, the complex node memory returned can be divided into two parts. Use the first part to store singly linked list data and second part is memory pool to serve some objects. The first part is represented as green and second part can be thought of SSDAM memory pool. Pool is divided into empty objects, and when they are empty, they are orange and when they are occupied/referred by some object they are blue similar to concept of SSDAM. In SSDAM-E, instead of allocating 1000 objects in a single memory pool during one cycle, we will allocate 100 objects in each individual pool out of total number of objects (here 1000) we need to allocate in one cycle. So, we will have 1000/100 = 10 pools and size of each pool should be 100 ∗ sizeof(MyPracticalClass). These pools are connected using a singly linked list chain internally by memory manager using the complex singly node. Now, there are 10 complex nodes. For sake of simplicity, we are showing two complex nodes depicted in Figure 13.

After first 100 object allocations done by the SSDAM-E memory manager, the first complex node is exhausted and will look like as shown in Figure 14.

The free orange memory areas are now occupied by 100 objects, and thus, the memory manager turns blue. The next complex node in the chain will now be used linearly from left to right for the next 100 object allocations, filling all 100 memory locations that its pool can provide. Figure 15 shows a singly linked list with no free complex nodes.

The next 800 objects consume the next 8 complex nodes. So, after 1000 allocations (done in one cycle out of 500000)

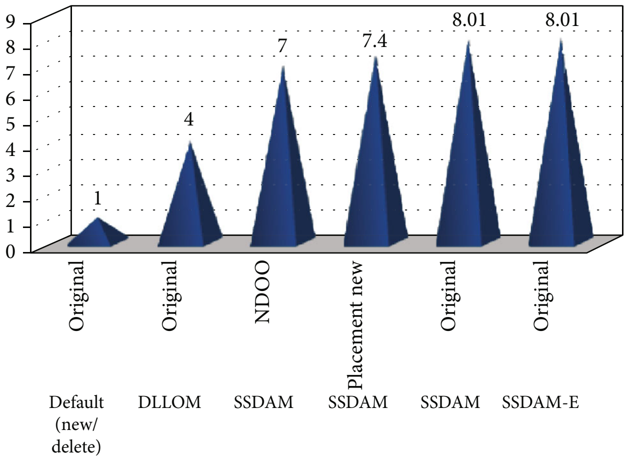

Figure 17: Comparison of execution time of different methods.

the SSDAM-E is full. The next phase is deallocation of 1000 objects in the same cycle. This is done by calling MemoryManager::freeAddress() with memory address of object as parameter, i.e., pointer to the object. After 1000 deallocations, all 10 complex nodes will have their memory pool object memories freed as shown in Figure 16.

Now, 1 cycle (out of 500000 cycles) of 1000 allocations and deallocations is done. In the next cycle, the memory manager's 10 complex nodes' pool memory can be reused to do the next 1000 allocations and deallocations similar to the process described above. In SSDAM-E, we can have two cases: the best case and the worst case. In the best case, we have a memory pool of 1000/1000 = 1, i.e., we will have only one pool of memory of size 1000. In the worst case, the number of memory pools will be 1000/1 = 1000, i.e., we have 1000 memory pools of size 1.

5.3.3. Benchmarks. The best case of SSDAM-E gave the average running time of less than 3.800 seconds which is 2.7-5% faster than SSDAM. The worst-case SSDAM-E gave the average running time of 5.650 sec which is faster than DLLOM average time of 7.496 sec.

5.3.4. Pseudocode

6. Results and Discussion

The following situations were considered and compiled to get the results.

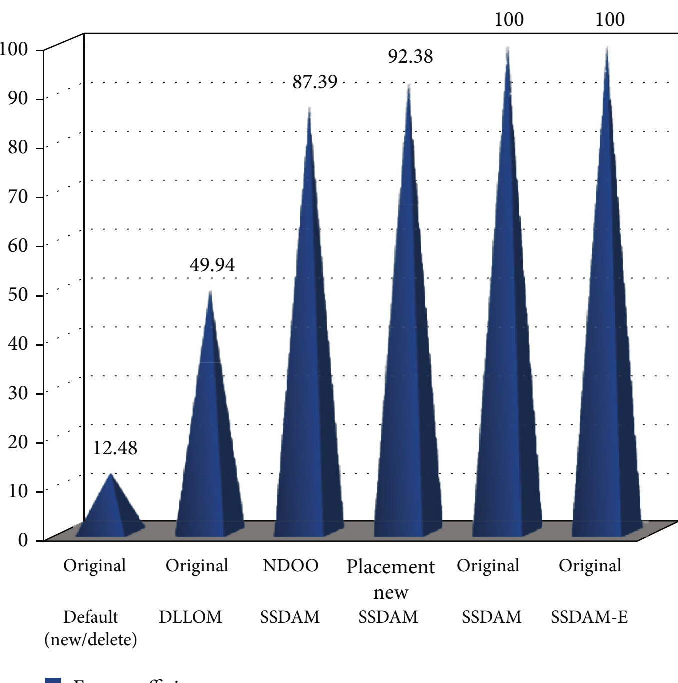

Figure 18: Comparison of energy efficiency of different methods.

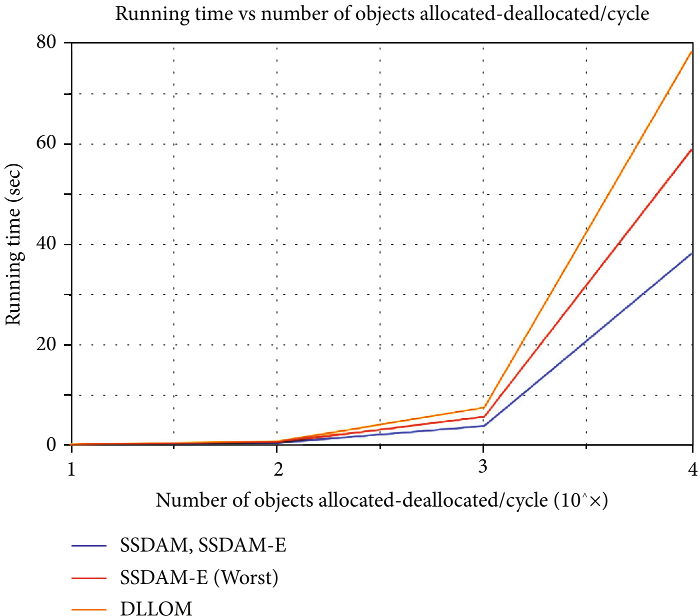

Figure 19: Running time vs. number of objects allocated/deallocated per cycle.

(1) Allocate a large amount of memory of a single size, then free it all

(2) Allocate a small amount of memory, free it, and repeat this loop several times

(3) Allocate lots of memory of the same size, free half of it (e.g., the even allocations), and then allocate and free memory in a loop

(4) Allocate memory in parallel using multiple threads

Table 1 depicts the performance of various methods relative to default (new/delete) operator-based mechanism. We ran the code in a similar environment and hardware conditions as other memory allocators (refer to the General Code for all allocators section above). The best case of SSDAM-E gave an average running time of less than 3.800 seconds. SSDAM-E's best case performed 2.7–5% faster than SSDAM. Insights on this can be gathered from what Emery Berger said in the CppCon 2020 Keynote "Performance (Really) Matters": There is a lovely paper

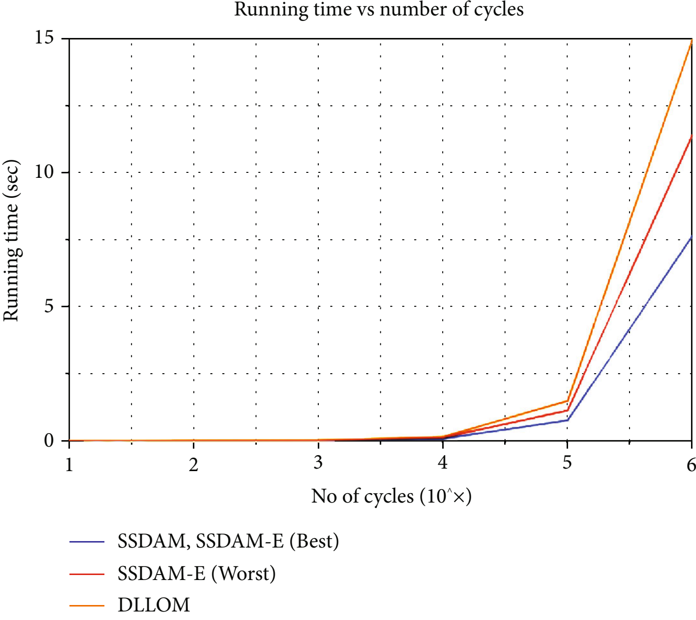

Figure 20: Running time vs. number of cycles.

that a few of my colleagues wrote back in 2009. So, we believe that cache efficiency is what makes SSDAM-E the best case. The worst case, SSDAM-E gave an average running time of 5.650 sec faster than the DLLOM average time of 7.496 sec. Figures 17 and 18 show time and energy consumption requirements of various methods.

Figures 19 and 20 depict the visual representation for running time against number of objects allocated/deallocated per cycle and number of cycles, respectively. We have discussed the implementations and trade-offs between them. We first performed a single benchmark test, keeping fixed the number of cycles of allocation-deallocation and the number of objects allocated in one cycle. In the case of SSDAM, we also added variations (in one case, we used the placement new operator rather than the class initialize function, and in the other case, we used new and delete operator overloading within our class, i.e., MyPracticalClass). Below graphs show the performance against each other when the computation at hand varies.

7. Conclusion

This article discusses various concepts and implementations of memory management techniques that can be used at the source code level and are designed to be pragmatic in their application. All of the approaches discussed far have a low runtime overhead and are thus applicable to a wide variety of use cases. In the future, we intend to investigate the integration aspects of the approaches discussed here and to attempt to apply them to existing systems that are in production and known to have memory performance issues. The inherent drawbacks of memory management operators are highlighted. The application of these operators is intended to be extremely generic, much like the concept of dynamic memory. As a result, they are unable to utilize the various optimization techniques and opportunities that particular use cases present. Each source code file is modeled after its own unique memory usage pattern, which speeds up memory management procedures. The SSDAM, SSDAM-E, and DLLOM strategies have been evaluated and compared to the performance of the new and delete operators. SSDAM-E, SSDAM with new delete operators, and DDLOM reduce memory usage by 8.01, 7.0, and 4.0 times, respectively. In the worst-case scenario, SSDAM-E provided an average execution time 5.650 seconds faster than DLLOM. As far as energy efficiency is concerned, SSDAM-Original and SSDAM-E-Original achieve 100 percent, whereas new/delete operators have a baseline efficiency of 12.48 percent.

Data Availability

Data and code are available with authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] E. Berger, K. McKinley, R. Blumofe, and P. Wilson, “Hoard: a scalable memory allocator for multithreaded applications,” in *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, 2000.

[2] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, “Learning-based memory allocation for C++ server workloads,” in *the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[3] M. Neely, *An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation*, MS Thesis Department of Computer Science, Univ. of Colorado, Boulder, 1996.

[4] D. Lea, *A Memory Allocator*http://g.oswego.edu/dl/html/malloc.html.

[5] P. R. Wilson, M. S. Johnston, M. Neely, and D. Boles, "Dynamic storage allocation a survey and critical review," Technical Report, Department of Computer Science, Univ. of Texas, Austin, TX, 1995.

[6] B. Zorn and D. Grunwald, "Empirical measurements of six allocation intensive C programs," Technical Report CU-CS-604-92, Department of Computer Science, Univ. of Colorado, Boulder, CO, 1992.

[7] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Building high performance custom and general-purpose memory allocators," in *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp. 114–124, 2000.

[8] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *6th International Conference on Learning Representations*, Vancouver, BC, Canada, 2018.

[9] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: solved," in *Proceedings of the 1st International Symposium on Memory Management (ISMM '98)*, pp. 26–36, ACM, New York, NY, USA, 1998.

[10] D. Häggander and L. Lundberg, "Attacking the dynamic memory problem for SMPs," in *Proc. of the 13th International Conf. on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, USA, 2000.

[11] J. M. Robson, "Worst case fragmentation of first fit and best fit storage allocation strategies," *The Computer Journal*, vol. 20, 1977.

[12] N. Cohen and E. Petrank, "Limitations of partial compaction: towards practical bounds," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI13)*, pp. 309–320, ACM, New York, USA, 2013.

[13] N. Cohen and E. Petrank, "Limitations of partial compaction," *ACM Transactions on Programming Languages and Systems*, vol. 39, no. 1, pp. 1–44, 2017.

[14] S. Ghemawat and P. Menage, *Google*, TCMalloc, 2020, https://github.com/google/tcmalloc.

[15] *Google*, 2020, pprof. https://github.com/google/pprof.

[16] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google Vizier: a service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1487–1495, 2017.

[17] R. R. Fenichel and J. C. Yochelson, "A LISP garbage-collector for virtual-memory computer systems," *Communications of the ACM*, vol. 12, no. 11, pp. 611-612, 1969.

[18] W. J. Hansen, "Compact list representation," *Communications of the ACM*, vol. 12, no. 9, pp. 499–507, 1969.

[19] A. Bily, *Modern Garbage Collector for Hash Link and Its Formal Verification*, Meng Individual Project, Imperial College London, 2020.

[20] J. Coppeard, *Compacting Garbage Collection in Spider Monkey*https://mzl.la/2rntQlY.

[21] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write- rationing garbage collection for hybrid memories," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, pp. 62–77, 2018.

[22] M. Wu, Z. Zhao, H. Li et al., "Espresso: brewing Java for more non-volatility," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, pp. 70–83, 2018.

[23] B. Daloze, S. Marr, and D. Bonetta, "Efficient and thread-safe objects for dynamically-typed languages," in *ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2016.

[24] D. Detlefs, A. Dosser, and B. Zorn, "Memory allocation costs in large C and C++ programs," *Software – Practice and Experience*, vol. 24, no. 6, pp. 527–542, 1994.

[25] B. Fitzpatrick, *Memcached*, 2014, http://memcached.org/.

[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS. ACM, 2012.

[27] U. Farooq and Z. Zhao, "Runtimedroid: restarting-free runtime change handling for android apps," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 110–122, 2018.

[28] D. Häggander and L. Lundberg, "Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor," in *Proc. of the 27th International Conf. on Parallel Processing*, Minneapolis, USA, 1998.

[29] C. M. Sharma, R. Porwal, and D. Sharma, "Testing object oriented software: issues, state-of-the-art and future," *International Journal of Computer Applications*, vol. 975, p. 8887, 2013.

[30] C. M. Sharma, A. K. S. Kushwaha, S. Nigam, and A. Khare, "On human activity recognition in video sequences," in *2011 2nd international conference on computer and communication technology (ICCCT-2011)*, pp. 152–158, 2011.

[31] M. K. Singh, S. Kumar, G. Bhatnagar et al., "A blend of analytical and numerical methods to compute orthogonal image moments over a unit disk," *Wireless Communications and Mobile Computing*, vol. 2022, Article ID 1344584, 15 pages, 2022.

[32] C. M. Sharma and S. K. Dinkar, "A survey on evolutionary clustering algorithms and applications," in *applications of advanced optimization techniques in industrial engineering*, pp. 23–34, CRC Press, 2022.

[33] K. S. Sundari and R. Narmadha, "Optimal energy efficient, load aware memory management system on SoC's for industrial automation," *Applied nanoscience*, 2022.

[34] I. A. Astrakhantseva, R. G. Astrakhantsev, and A. V. Mitin, "Randomized C/C++ dynamic memory allocator," in *journal of physics: conference series*, vol. 2001no. 1, IOP publishing, p. 012006, 2021.

[35] H. R. Aradhya, J. Fadnavis, and S. G. Gojanur, "Memory design and verification of SRAM-based energy efficient ternary content addressable memory," in *2021 5th international conference on information systems and computer networks (ISCON)*, pp. 1–7, IEEE, 2021.

[36] H. K. Liu, D. Chen, H. Jin et al., "A survey of non-volatile main memory technologies: state-of-the-arts, practices, and future directions," *Journal of Computer Science and Technology*, vol. 36, no. 1, pp. 4–32, 2021.