OSLAB Project 03

YaoShunyu LvYingzhe

10152130255 LvYingzhe

November 26, 2017

Basic Conceptions

- Maintain an ordered free list
- De-fragmentation after free

Metadata

```
typedef struct _Node {
    struct _Node *next;
    int size;
} Node;

typedef struct _Header {
    int size;
    struct _Header *magic;
} Header;
```

How Allocation Works?

- Verify argument
- Round up with bitwise operation
- Find a suitable block
- Split/Remove from free list
- Transform into Header and return

Split or Remove?

Condition	Action
pNode→size $\geq$ size + sizeof(Node)	Split
pNode→size $<$ size + sizeof(Node)	Remove

How Free Works?

- Verify pointer argument
- Transform into Node
- Insert into free list
- Try to merge with left/right neighbour

Dilemma

16-byte header can store mere information!

Free list? Fully connected? We want both!

Design

- 24-byte Header
- Maintain a free list
- Link between neighbours

24-byte Header

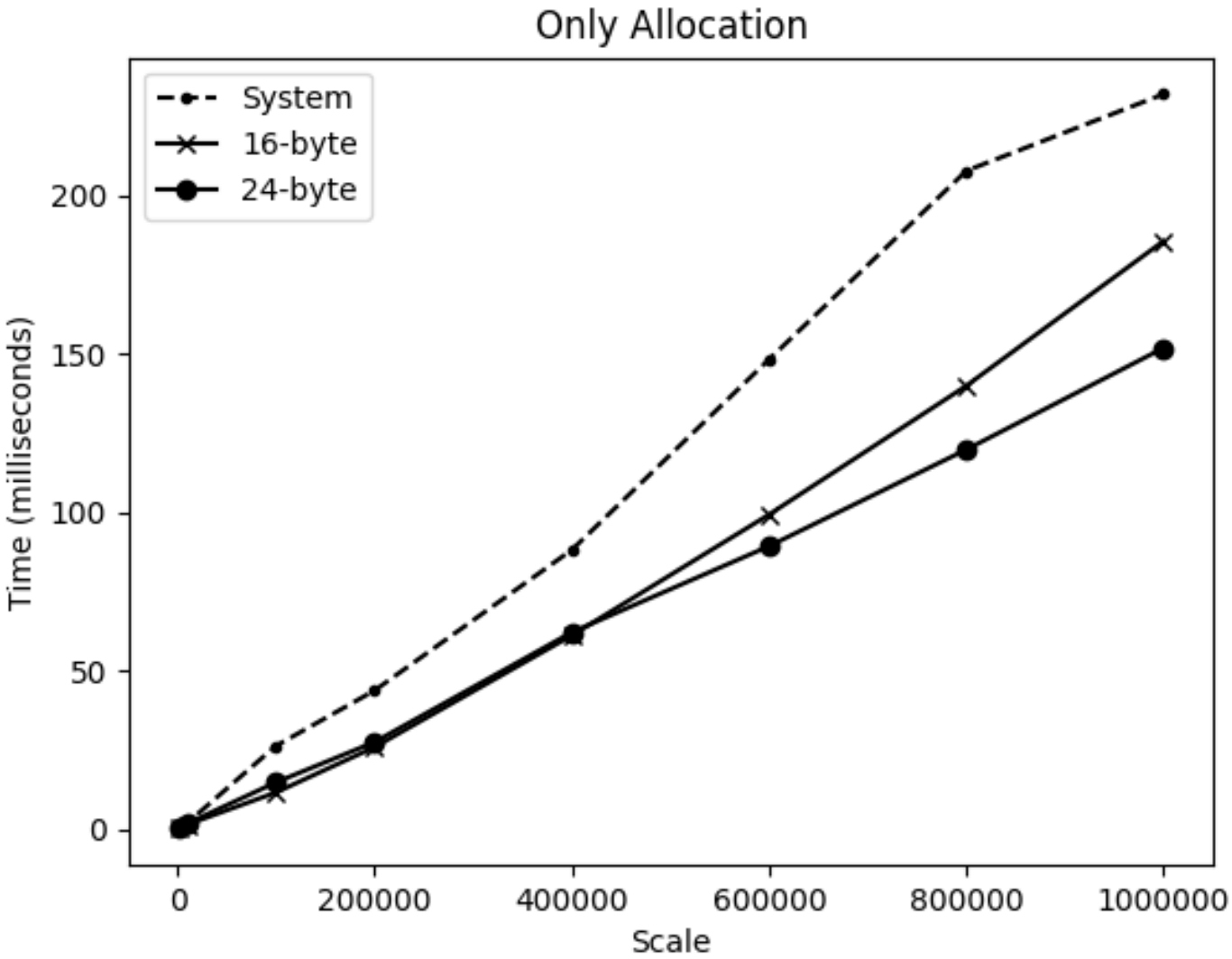
```
typedef struct _Node {
    struct _Node *free_next;
    struct _Node *pre;
    int size;
} Node;
```

Improvement

- Same allocation speed
- Much faster free speed

Allocation without Free

```
for i <- 1 to N:
        ptr[i] = alloc(rand(1, BOUND), STYLE)
```

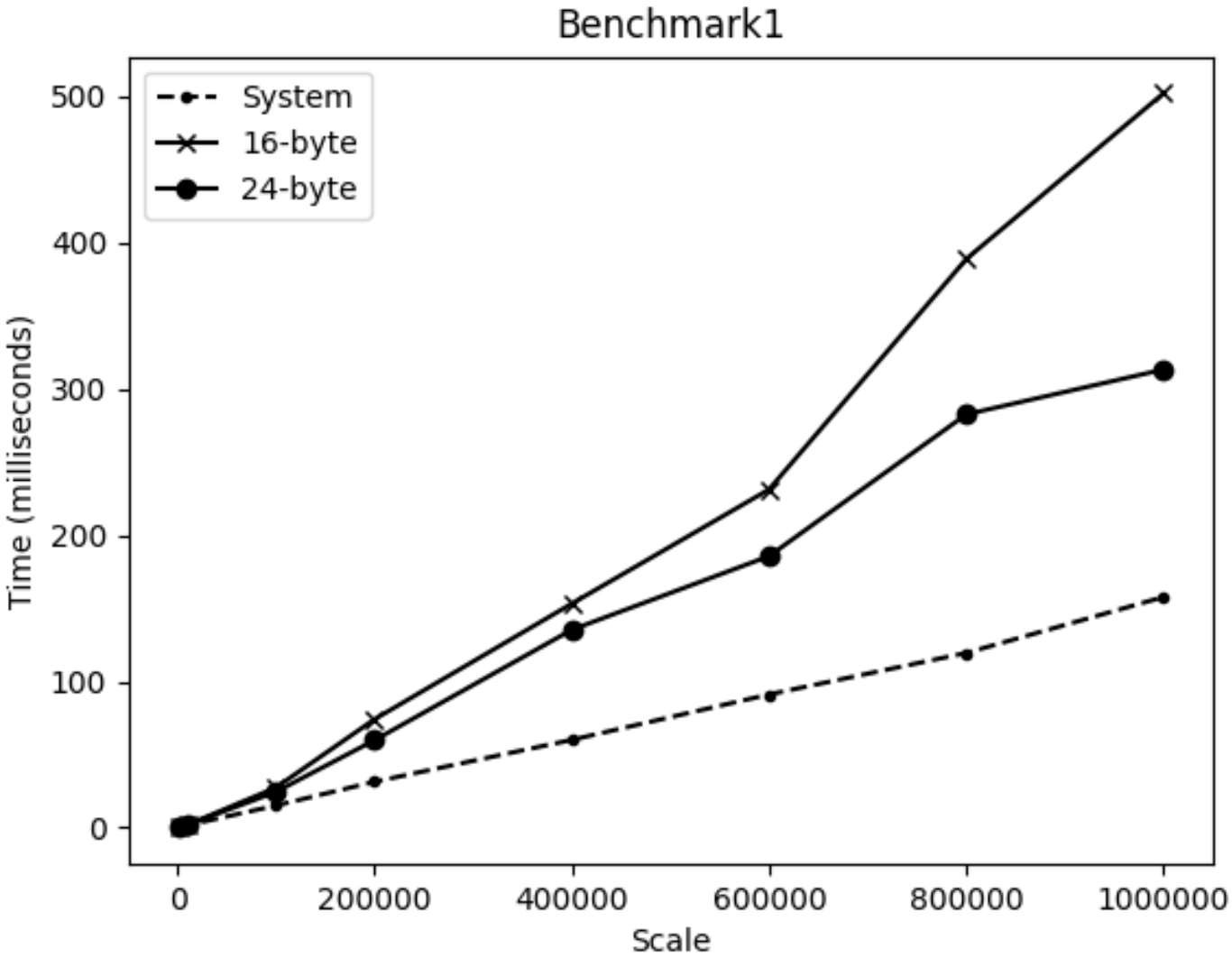

Only Allocation
System
16-byte
24-byte
Time (milliseconds)
0
50
100
150
200
0
200000
400000
600000
800000
1000000
Scale

Conclusion

The allocation is not the bottleneck.

Random Allocation/Free Calls

```
op <- rand(ALLOC, FREE)
if op = ALLOC:
        size <- rand(1, BOUND)
        list.add(alloc(size))
else if op = FREE and list.size() > 0:
        index <- rand(list.size())
        free(list[index])
        list.remove(index)
```


Benchmark1
System
16-byte
24-byte
Time (milliseconds)
0
100
200
300
400
500
0
200000
400000
600000
800000
1000000
Scale

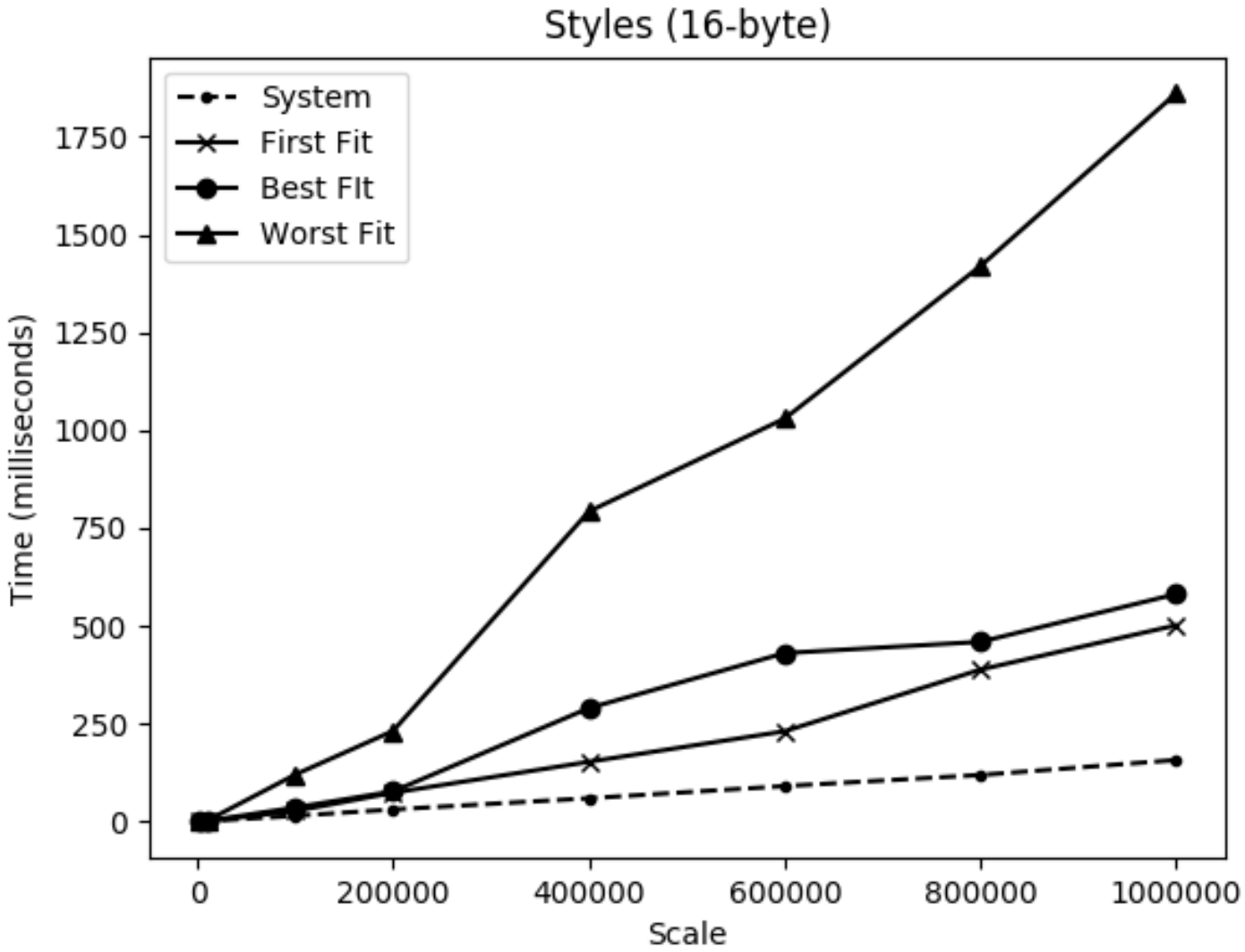
Conclusion

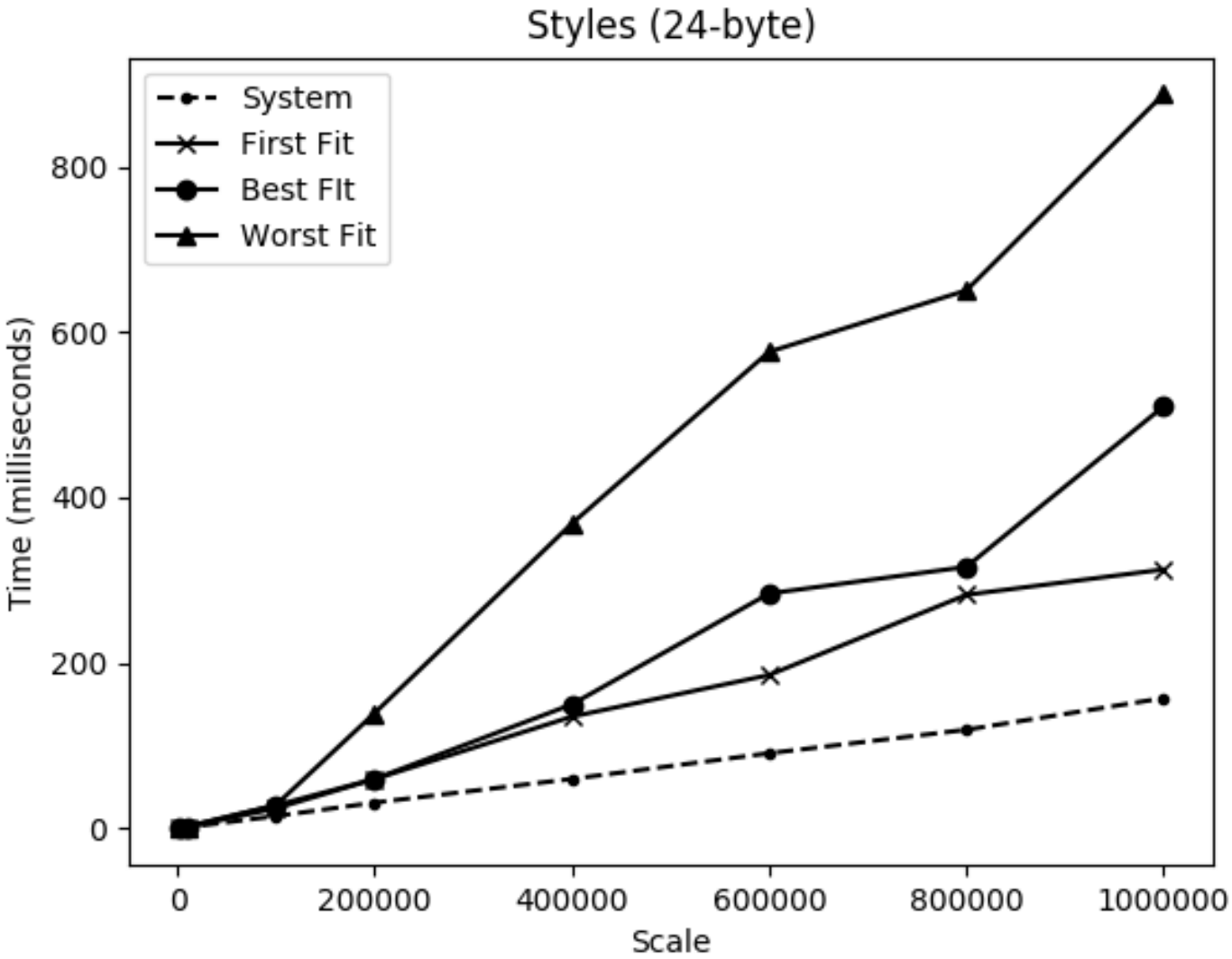
Both 16-byte and 24-byte stand the violent test, despite that system performs better.

Different Allocation Styles

Same test method as short free list

Change allocation styles respectively


Styles (16-byte)
System
First Fit
Best FIt
Worst Fit
Time (milliseconds)
0
250
500
750
1000
1250
1500
1750
0
200000
400000
600000
800000
1000000
Scale


Styles (24-byte)
System
First Fit
Best FIt
Worst Fit
Time (milliseconds)
0
200
400
600
800
0
200000
400000
600000
800000
1000000
Scale

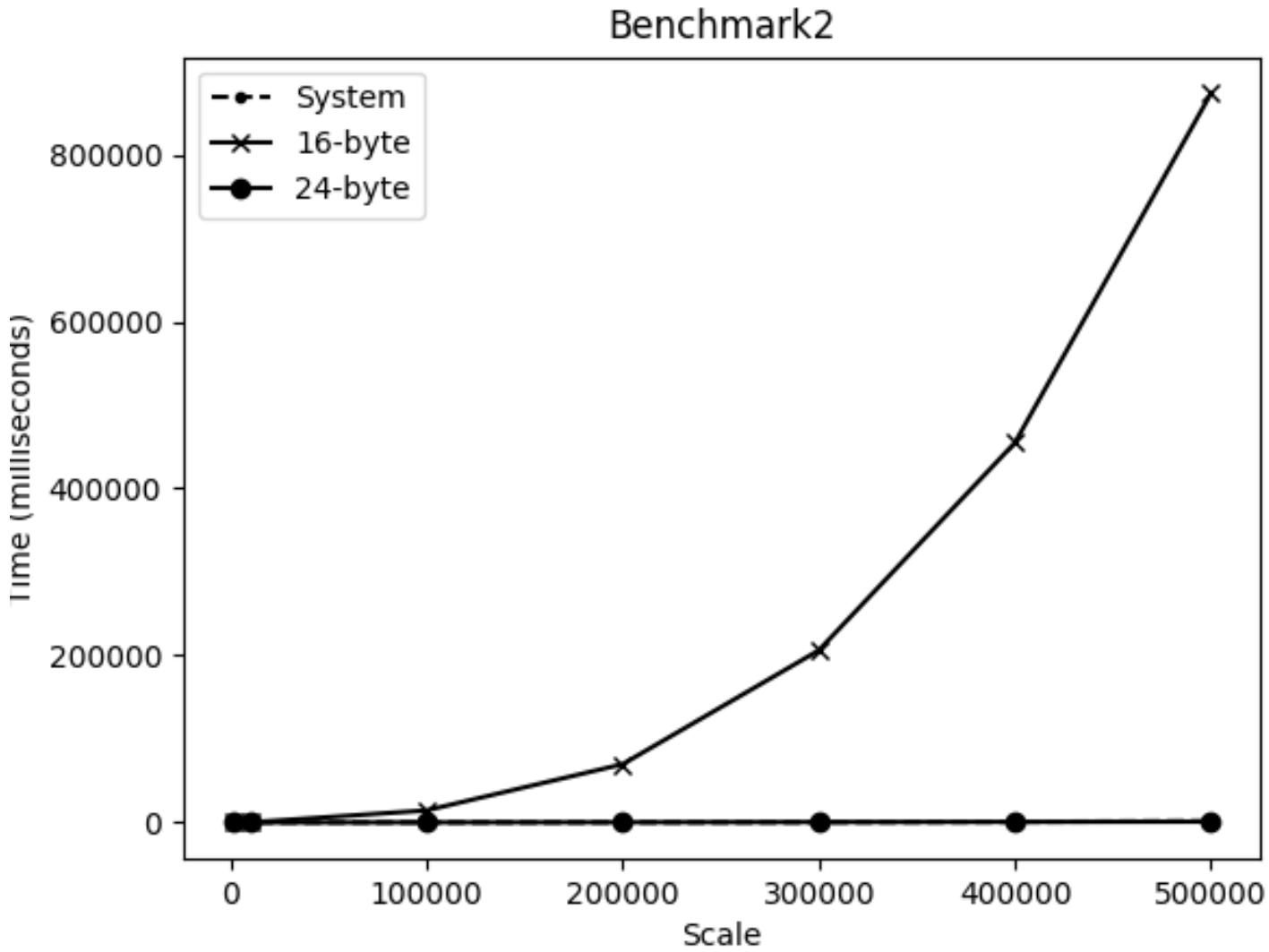
Conclusion

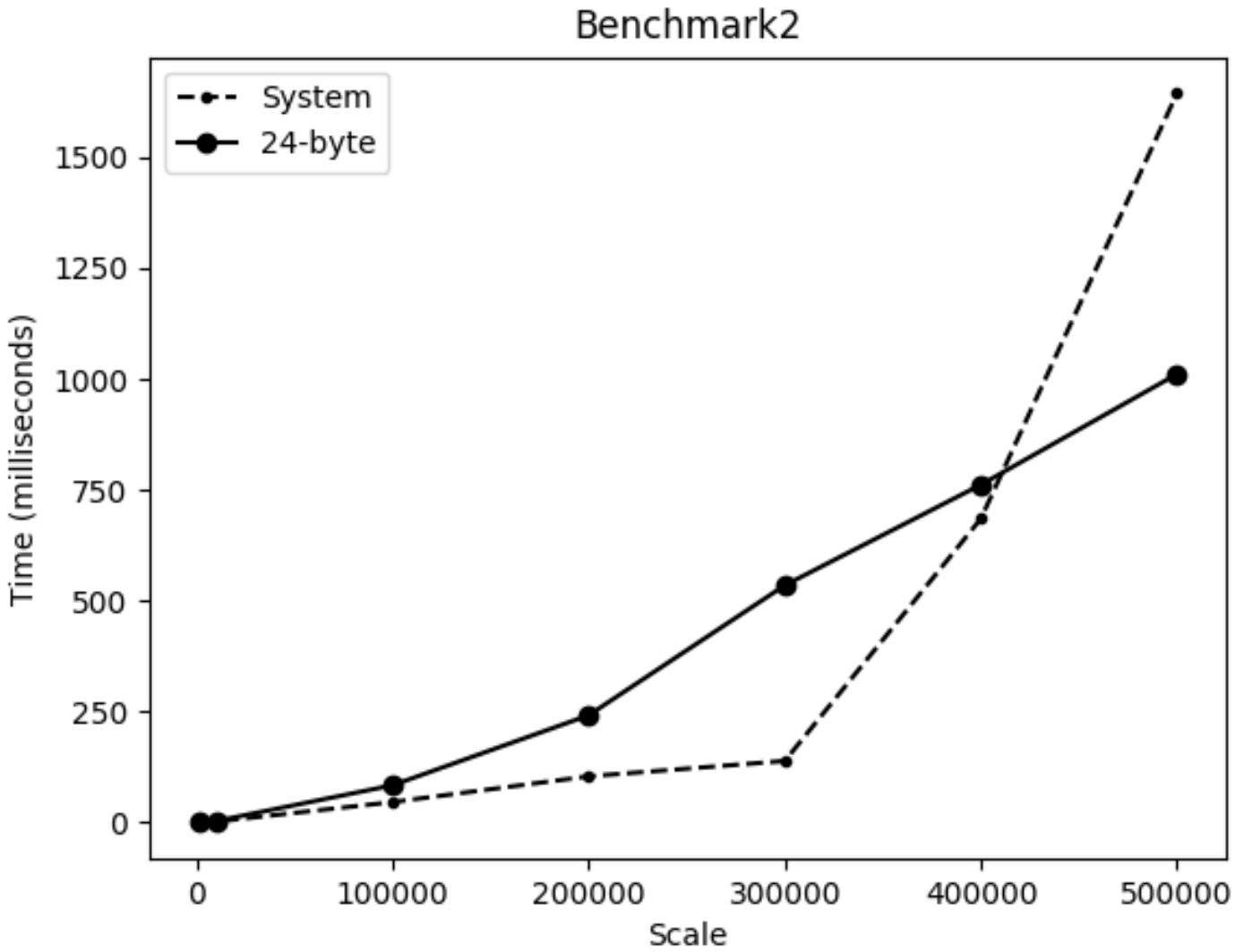
Performance:

First Fit $>$ Best Fit $>$ Worst Fit

Numerous Free after Allocation

```
indexes <- Fisher_Yates(N)
for i <- 1 to N:
        ptr[i] = alloc(rand(1, BOUND))
for i <- 1 to N:
        free(ptr[indexes[i]])
```


Benchmark2
System
16-byte
24-byte
800000
600000
400000
200000
0
Time (milliseconds)
0
100000
200000
300000
400000
500000
Scale


Benchmark2
System
24-byte
Time (milliseconds)
0
250
500
750
1000
1250
1500
0
100000
200000
300000
400000
500000
Scale

Conclusion

When free list is long,
the cost of free for 16-byte version is significant,
while 24-byte version performs like system malloc.

Comparison

Method	Speed	Fragment	Application
Dynamic Allocator	Moderate	Little External No Internal	malloc
Buddy Allocator	Fast	Little External Much Internal	jelloc
Slab Allocator	Very Fast	No External Much Internal	kalloc

Summary

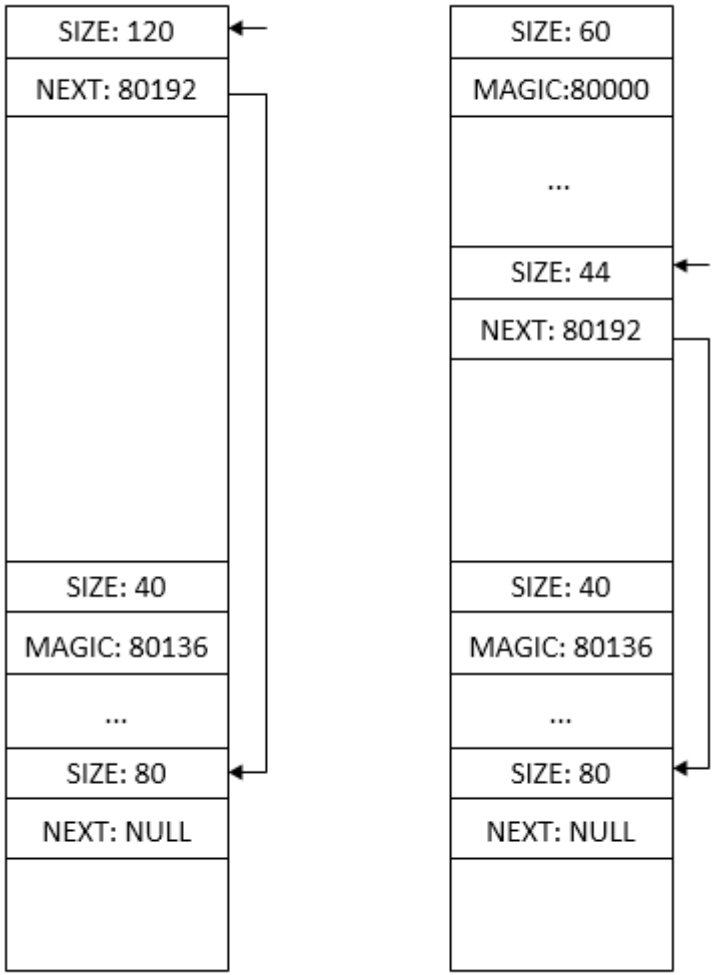
Balance the cost and gain is the eternal theme.

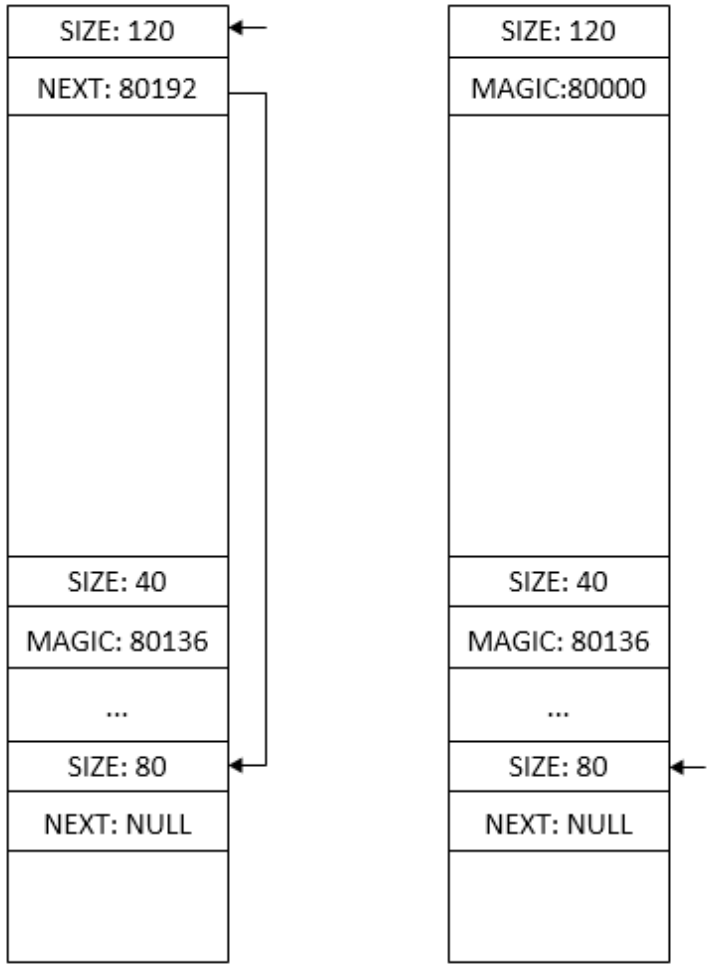
Nothing is best, only most suitable.

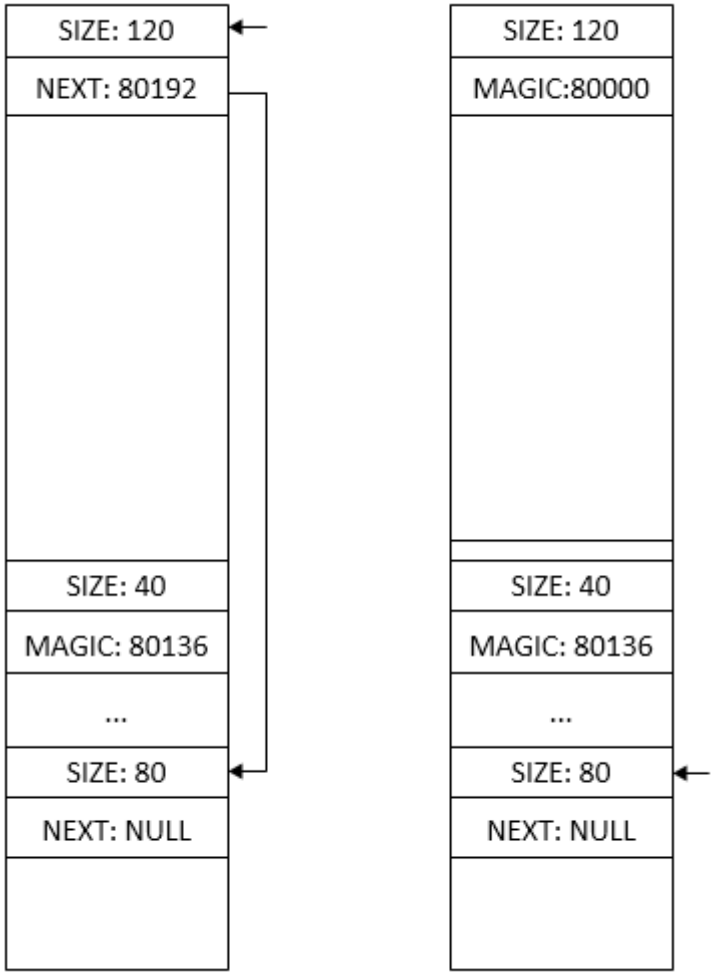
Fin

Thank you!

Condition	Action
pNode→size $\geq$ size+sizeof(Node)	Split
pNode→size == size + 8	Remove with extra 8-byte
pNode→size == size	Remove


SIZE: 120
NEXT: 80192
SIZE: 40
MAGIC: 80136
...
SIZE: 80
NEXT: NULL
SIZE: 60
MAGIC:80000
...
SIZE: 44
NEXT: 80192
SIZE: 40
MAGIC: 80136
...
SIZE: 80
NEXT: NULL


SIZE: 120
NEXT: 80192
SIZE: 40
MAGIC: 80136
...
SIZE: 80
NEXT: NULL
SIZE: 120
MAGIC:80000
SIZE: 40
MAGIC: 80136
...
SIZE: 80
NEXT: NULL


SIZE: 120
NEXT: 80192
SIZE: 40
MAGIC: 80136
...
SIZE: 80
NEXT: NULL
SIZE: 120
MAGIC:80000
SIZE: 40
MAGIC: 80136
...
SIZE: 80
NEXT: NULL