

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/262375150>

A Comparative Study on Memory Allocators in Multicore and Multithreaded Applications – SBESC 2011 – Presentation Slides

Data · May 2014

CITATIONS

0

READS

274

4 authors, including:



Taís Ferreira

Universidade Federal de Uberlândia (UFU)

13 PUBLICATIONS 64 CITATIONS

[SEE PROFILE](#)



Rivalino Matias Jr.

Universidade Federal de Uberlândia (UFU)

131 PUBLICATIONS 784 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Reliability of Operating Systems [View project](#)

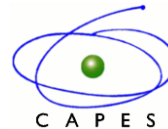


Theoretical and Experimental Analyses of Dynamic Memory Allocations [View project](#)

A Comparative Study on Memory Allocators in Multicore and Multithreaded Applications

Taís Borges Rivalino Matias Autran Macêdo Lúcio Borges

School of Computer Science & School of Mathematics
Federal University of Uberlândia
Uberlândia MG, Brazil



AGENDA

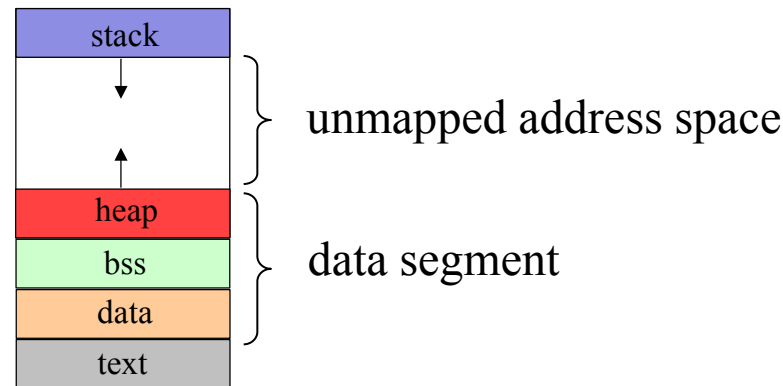
- Introduction
- User-level Memory Allocators
- Experimental Study
- Result Analysis
- Final Remarks

Introduction

- Dynamic memory allocations are one of the most ubiquitous operations in computer programs.
 - sophisticated real-world applications need to allocate/deallocate memory many times during their lifetime.
 - thus the performance of memory mgmt routines is very important, but it is frequently neglected in software design.
- A **memory allocator** is the code responsible for implementing the memory mgmt routines.

Memory Allocators

- A memory allocator manages the **heap** of application processes
- The **heap** is the region of the process address space used to meet requests for dynamic memory allocations (`malloc`, `new`, ...).



- Therefore, understanding how memory allocators work is very important to improve software performance

Memory Allocators (cont'd)

- There are two classes of memory allocators
 - User-Level MA (UMA)
 - it serves the requests from application processes
 - Kernel-Level MA (KMA)
 - it serves the requests from OS kernel subsystems
- When a memory request exceeds the available memory size in the process' **heap**, the UMA requests additional memory to the OS (the KMA).
 - this received portion of additional memory is then linked into the heap of the process and managed by its UMA.

Memory Allocators (cont'd)

- The UMA is an integral part of the applications
 - its code is usually stored in the standard C library (e.g., **glibc**)
 - **glibc** is automatically linked (statically or dynamically) to all programs, bringing in its default UMA.
- It is possible to use another UMA than that available in the glibc.
 - this is up to the programmer to choose the UMA of interest
 - however, many programmers don't know about this possibility
 - the transparent use of glibc hides these details

Memory Allocators (cont'd)

- Nowadays there are plenty of memory allocators:
 - hoard, jemalloc, nedmalloc, TCMalloc, TSLF, Dmalloc, ptmalloc, ...
- Each memory allocator algorithm has a different approach to manage the heap
- Experienced software engineers may decide to write a specific memory allocator for their application needs
 - e.g., Firefox, Chrome, PostgreSQL, Apache httpd, ...
- Currently, there are several proprietary and open source memory allocator algorithms
 - the choice of the allocator that provides the best performance for a given application must be based on experimental tests.

Motivation

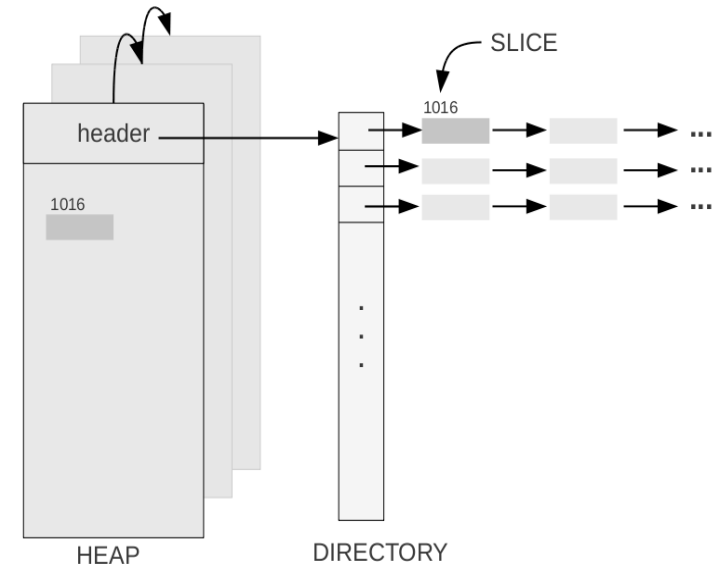
- Several studies have investigated the performance of UMA algorithms.
 - most of them are based on synthetic benchmarks (e.g., mtmalloc test)
 - they stress the UMA routines and data structures using random operations
- The above approach can hardly be generalized for real world applications.
- We present a study that experimentally compares seven UMA, using real applications and workloads.
 - the chosen applications are part of a high-performance stock trading middleware, which is composed of three main applications.
- Our motivation for this choice is that middleware applications in general have high demand for memory allocations
 - they also usually do not bring their own UMA, relying on the default allocator for portability purpose.

Background

- When a process calls `malloc/new` for the first time, the UMA requests to the OS a heap area.
 - it may require one or more heaps.
- Subsequently, it creates and initializes the heap header.
 - the header structures are practically the same for all today's memory allocators.

Background (cont'd)

- The Headers/Directories keep the list of free blocks
 - e.g., the memory slice 1016 is free.
- New heaps may be integrated to previous heaps.
- Although the allocators have many similarities in terms of data structures, they do differ considerably in terms of heap management.



General structures of an UMA

Background (cont'd)

- Each UMA implements a different approach to deal with the problems in this area, such as **blowup**, **false sharing**, and **memory contention**.
 - all these problems are strongly affected by multithreading and multiprocessing.
- **Blowup** is the consumption of the whole system memory by a process.
- **False sharing** is the phenomenon of two or more threads sharing the same cache line.
 - this occurs when two or more threads have acquired memory slices whose addresses are too close that they are located in the same cache line.
- **Memory contention** corresponds to the locking of threads due to a race condition for the same heap.
 - if multiple threads assigned to the same heap make memory requests, then contention will occur.

Background (cont'd)

- Each UMA deals with these problems using a different approach, which imposes a different performance among the allocators.
- For this reason we evaluate the performance of seven different memory allocators:
 - **Hoard** (3.8), **Ptmalloc** (2), **Ptmalloc** (3), **TCMalloc** (1.5), **Jemalloc** (2.0.1), **TLSF** (2.4.6), **Miser** (cilk_8503-i686);
- We selected these allocators because
 - their source code are available allowing us to investigate their algorithms and also they are widely adopted.

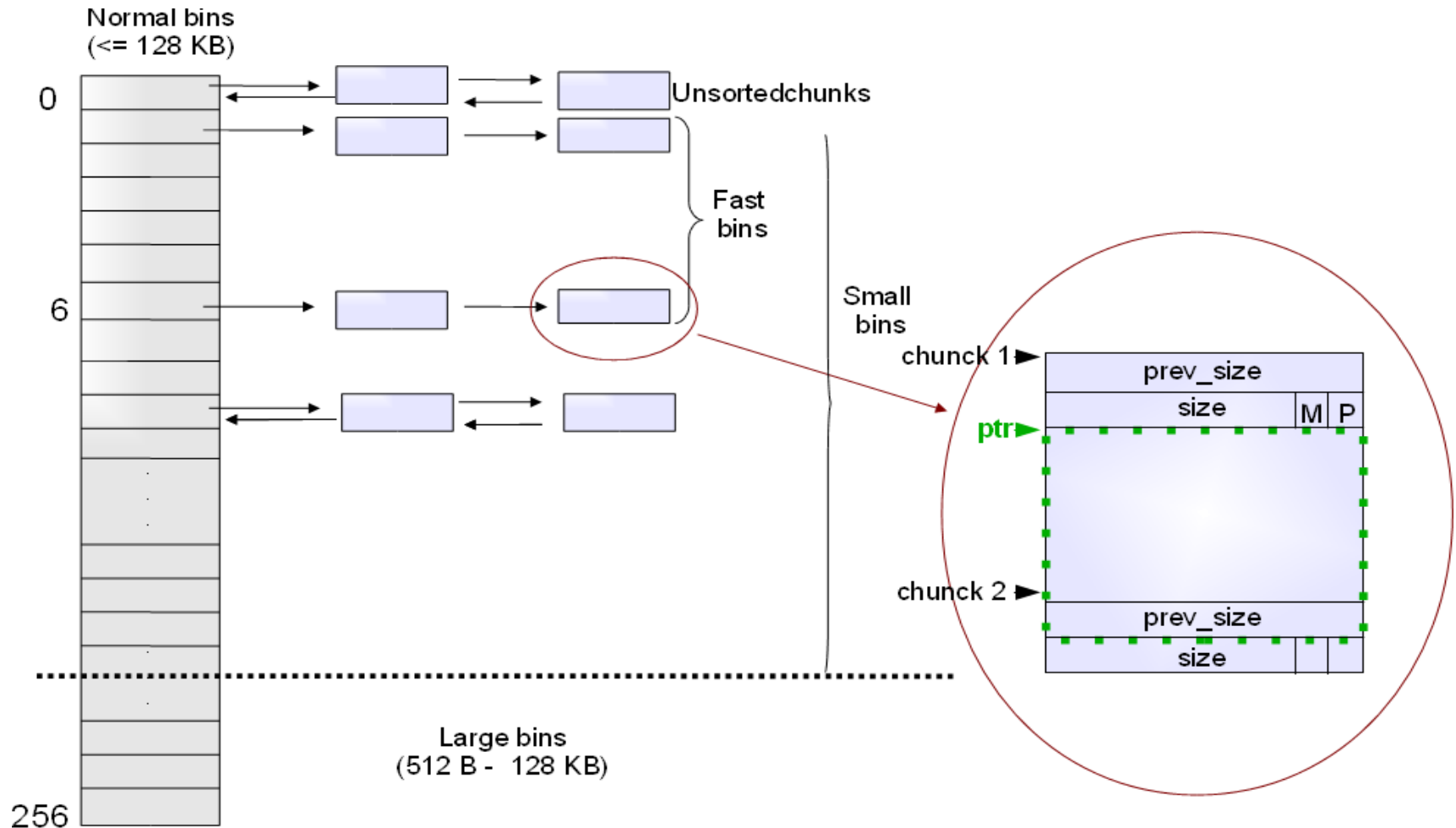
UMA Internals

- To show how an UMA works, we select the memory allocator currently embedded in **glibc**
 - this means that all applications running under Linux, and without their specific implementation of UMA, use it.
- This allocator is the *ptmalloc* (version 2)
 - *ptmallocv2* is based on another popular allocator called DLMalloc
- The design of *ptmallocv2* is focused on
 - multithreaded applications running on multiprocessor computers

UMA Internals (ptmallocv2)

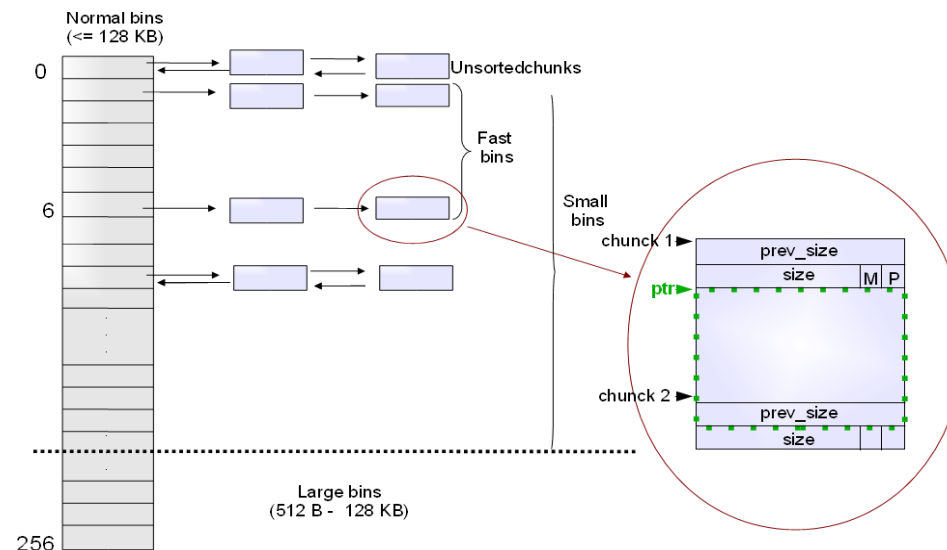
- Several UMA's don't show good performance for multithreaded applications
 - because the contention caused by multiple threads trying to access the same memory area (the Heap !)
- ptmallocv2 implements multiple heap areas (aka "Arenas") to reduce contention in multithreaded applications
 - whenever a thread requests a memory block and all arenas are in use (locked by other threads), a new arena is created
 - as soon as the new request is served, the other threads can also share the recently created arena.

UMA Internals (ptmallocv2) (cont'd)



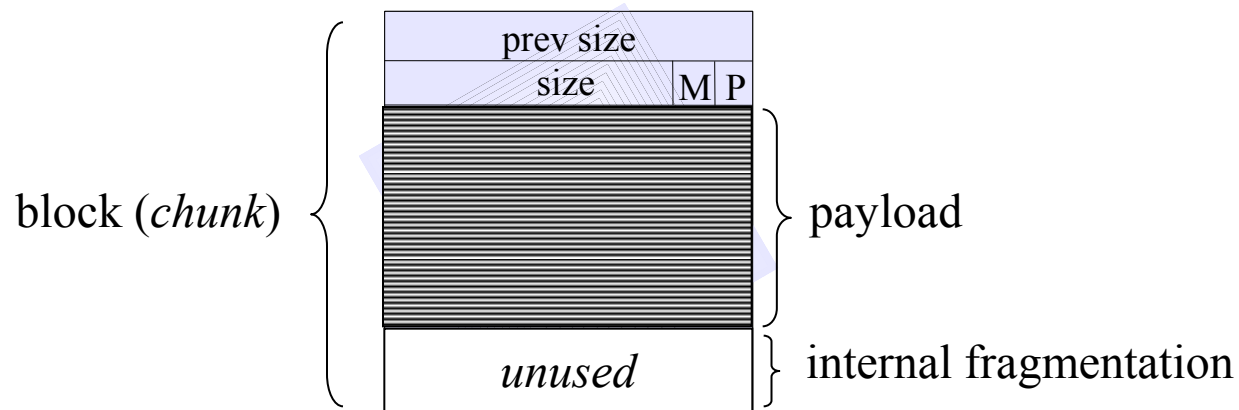
UMA Internals (ptmallocv2) (cont'd)

- When the application frees a chunk, ptmallocv2 puts it back in one of these lists.
- There are 2 classes of lists according to the chunk sizes :
 - Small bins (chunks of 16 – 512 bytes)
 - Large bins (up to 128 Kbytes)



UMA Internals (ptmallocv2) (cont'd)


- The ptmallocv2 provides memory chunks whose sizes are in power of two, starting from 16 (2^4) bytes.
 - if an application requests 20 bytes, ptmallocv2 provides a chunk of 32 bytes (2^5)
 - this is the smaller chunk size (in power of two) that fits the request
 - note that **12 bytes** are unused, which leads to **internal fragmentation**



UMA Internals (ptmallocv2) (cont'd)

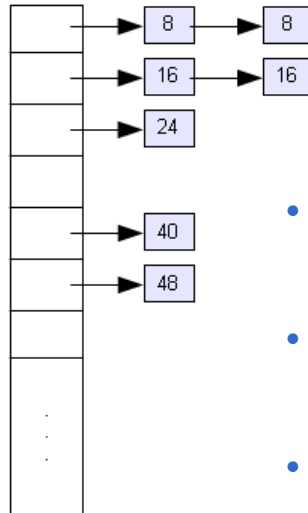
- External fragmentation also occurs
 - when there are non-contiguous free chunks to satisfy the request, although the heap area has enough space

UMA Internals (ptmallocv2) (cont'd)

 free memory



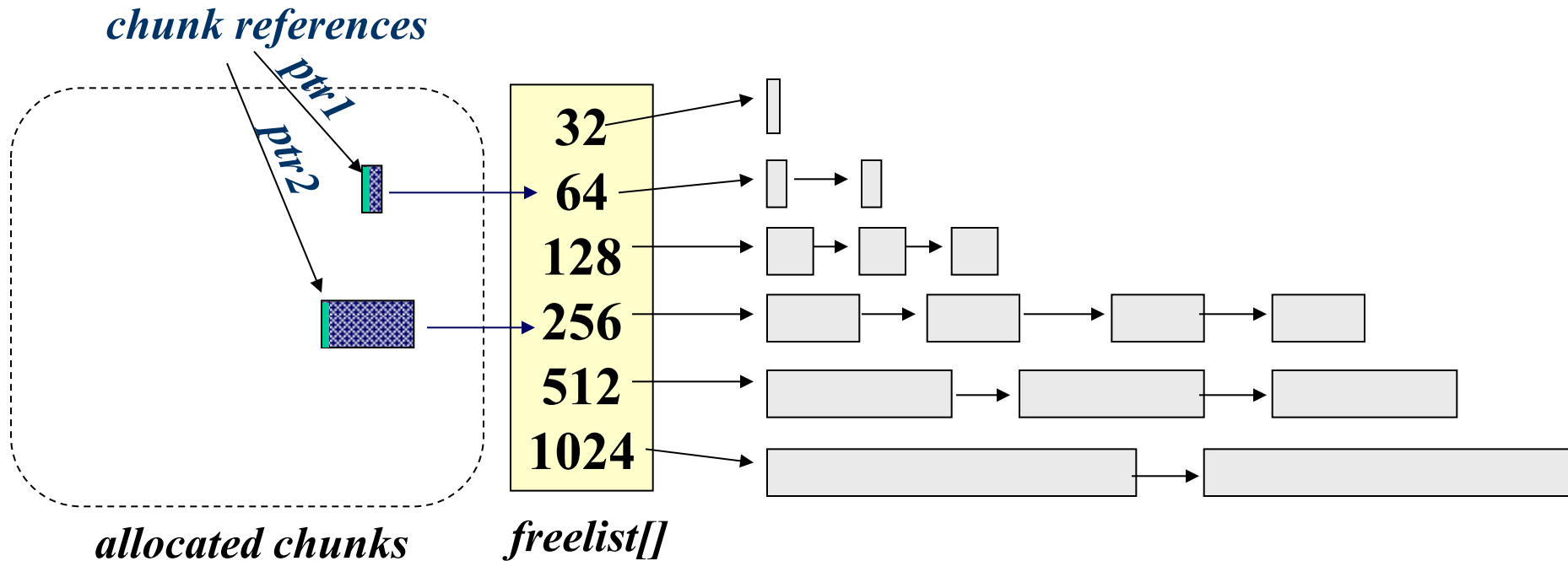
Free Lists



- If the application requests 55 bytes, ptmallocv2 can not provide this chunk from the current heap.
- although there are 160 bytes of free chunks, none of these chunks are large enough
- In this case ptmallocv2 should request a new memory chunk to the operating system
 - **it is significantly slower than using the process' heap area.**

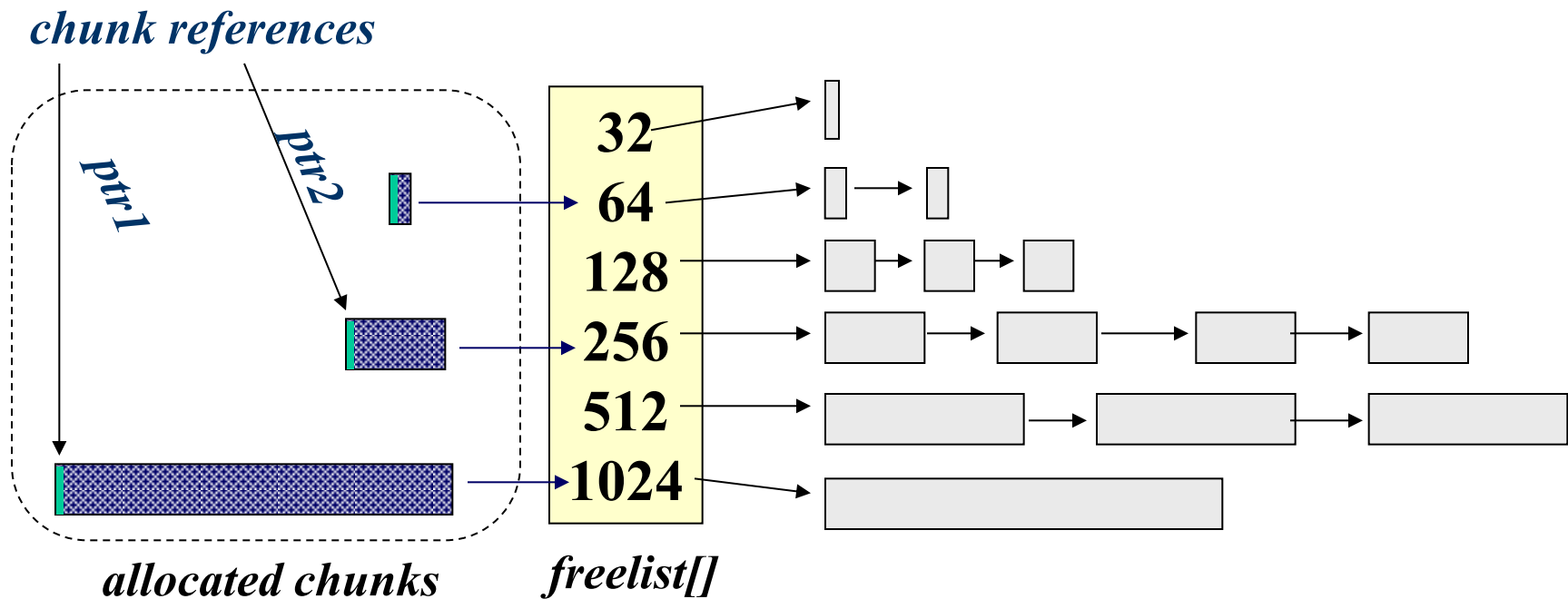
UMA Internals (ptmallocv2) (cont'd)

- This UMA is also vulnerable to memory leak
 - caused by programming mistakes



UMA Internals (ptmallocv2) (cont'd)

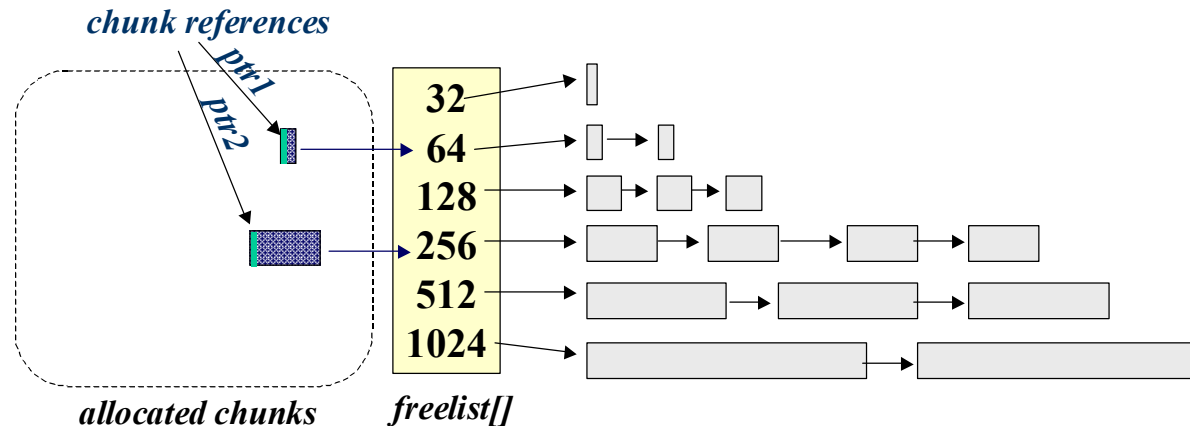
- This UMA is also vulnerable to memory leak
 - caused by programming mistakes



UMA Internals (ptmallocv2) (cont'd)

```
...
ptr1=malloc(60)
ptr2=malloc(250)
...
...
...
...
...

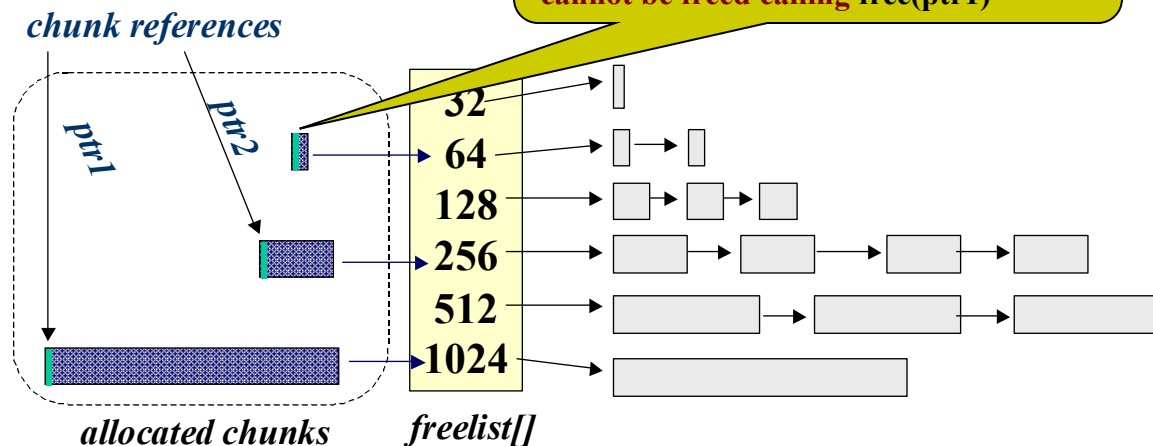
```



Leaked chunk !!!
 Since this is not referenced anymore, it
 cannot be freed calling free(ptr1)

```
ptr1=malloc(1000)

```



Summary of the Evaluated UMA

UMA	Complexity	Blowup	False Sharing	Contention
Hoard	$O(n)$	yes	yes	yes
Ptmallocv2	$O(1)$; $O(n)$	no	no	yes
Ptmallocv3	$O(1)$; $O(\log n)$	no	no	yes
TCMalloc	$O(1)$	yes	no	yes
Jemalloc	$O(1)$; $O(\log n)$	yes	no	yes
Miser	$O(1)$	no	no	no
TLSF	$O(1)$	yes	yes	yes

Methodology

- Our experimental study is composed of two phases.
- **Characterization of application memory usage**
 - we firstly ran a set of tests to characterize the memory usage of each investigated middleware applications.
 - it is very important to understand the memory allocation pattern for each tested application, since each allocator has a specific approach to deal with different request sizes
- **Evaluate UMA performance for each application**
 - we linked the three applications to each investigated allocator and analyzed their performance in terms of:
 - transaction (selling/buying order) rate
 - memory consumption
 - memory fragmentation

Methodology (cont'd)

- All tests were executed varying the number of processor cores
 - we tested for 1, 2, 3, and 4 cores
- All tests were replicated 15 times and used the averaged values to reduce the influence of experimental errors.

Instrumentation

- The middleware used is composed of three major applications.
 - App1 is responsible for the middleware core services (e.g., FIX protocol communication).
 - App2 is the session manager controlling all user sessions.
 - App3 is the data manager responsible for all transactional control, keeping track of each order flow and implementing all business rules (e.g., risk analysis).
- The whole middleware runs under Linux and its applications are implemented as multithreaded processes.

Instrumentation (cont'd)

- To characterize the memory usage for each application, we use the glibc memory allocation hook mechanism [14]
 - it let's to modify the behavior of `malloc/free`, `new/delete`, and `realloc` standard operations.
 - we install our own data collection routines using these hooks to keep track of all allocation and deallocation operations, for each application.

Instrumentation (cont'd)

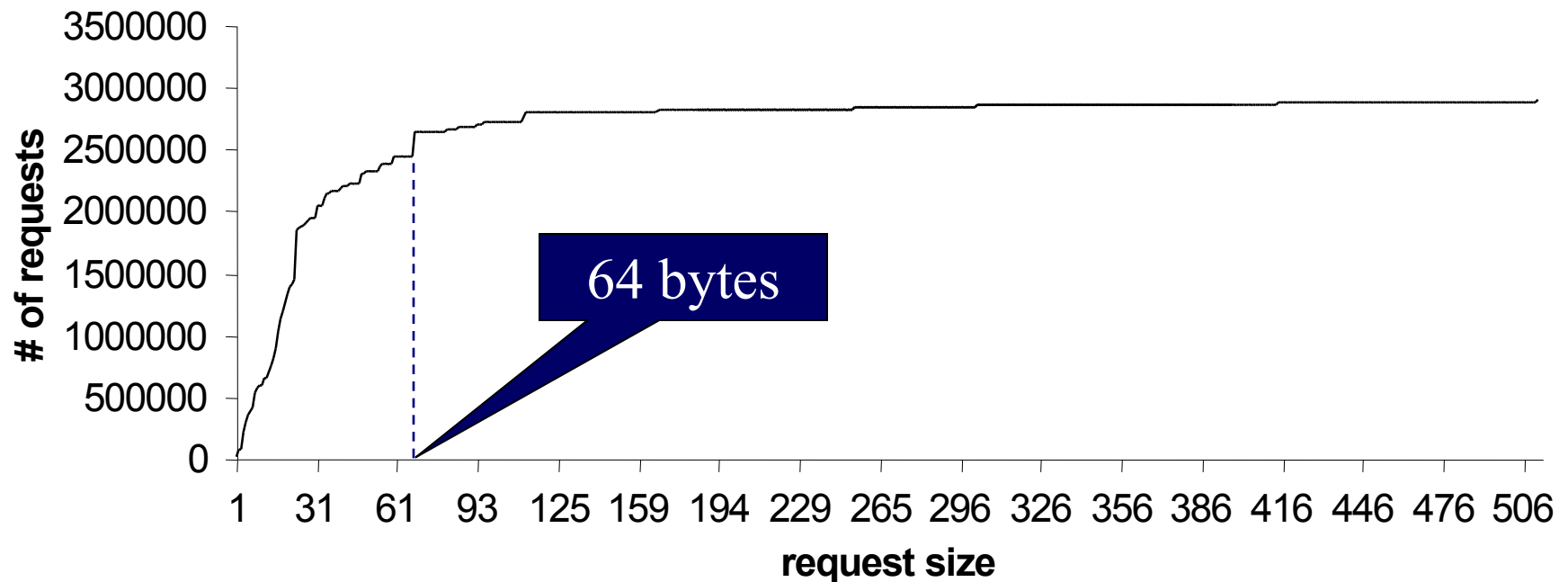
- We replace the default UMA of each application for each one of the seven evaluated allocators.
 - we instruct the Linux dynamic linker to load each evaluated allocator before we start a test.
 - this is accomplished by exporting the LD_PRELOAD environment variable [15].
 - `$ export LD_PRELOAD = libjemalloc.so; ./middleware_start`
 - it ensures all middleware applications are linked dynamically to the Jemalloc allocator.

Instrumentation (cont'd)

- We monitor the Linux RSS (*resident set size*) variable for each application process.
- We also use Kernel instrumentation, SystemTap [16], to monitor the number of kernel events related to memory fragmentation.
- To the best of our knowledge, none of the related experimental works have considered memory fragmentation in their studies.

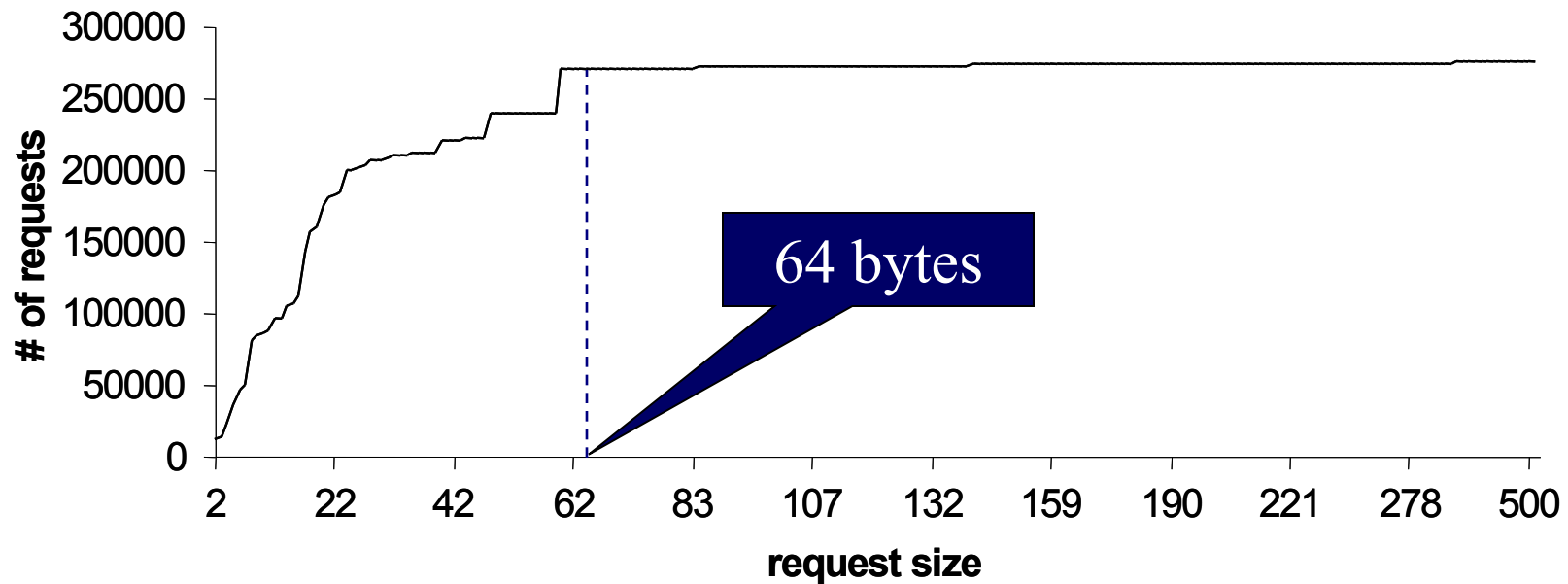
Memory Usage Characterization

Cumulative Distribution of Allocation Sizes for App1



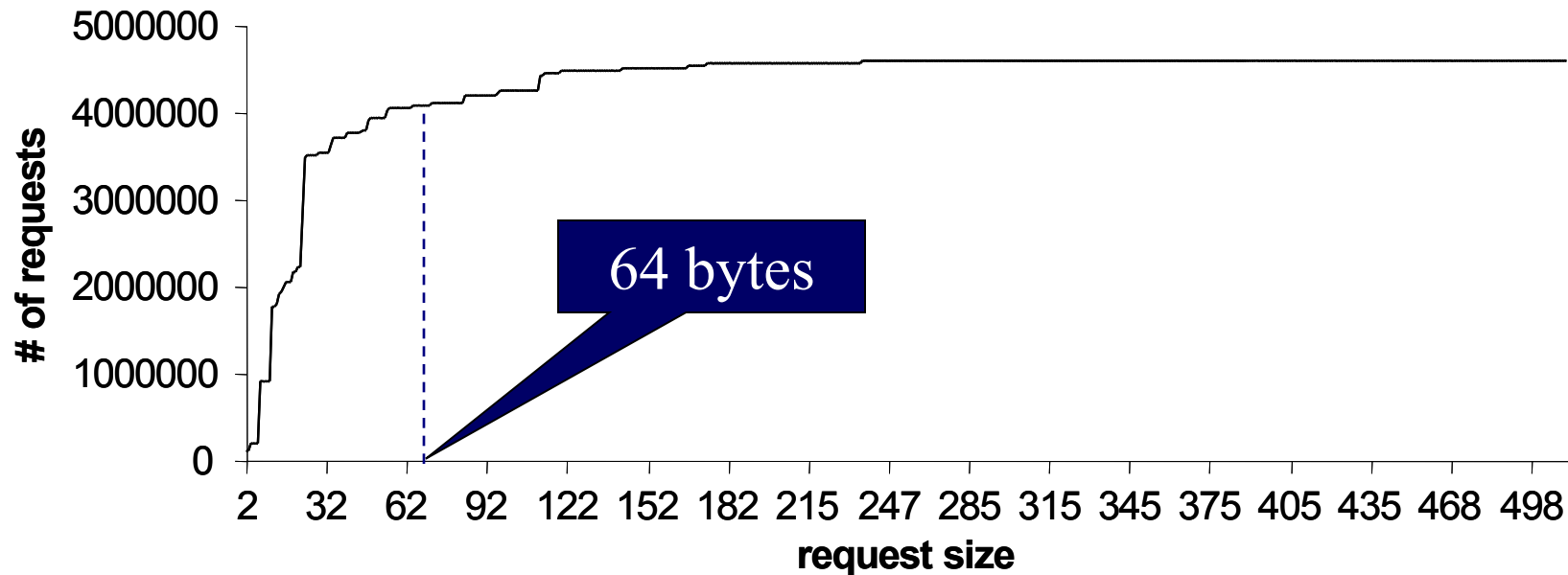
Memory Usage Characterization

Cumulative Distribution of Allocation Sizes for App2



Memory Usage Characterization

Cumulative Distribution of Allocation Sizes for App3



Memory Usage Characterization

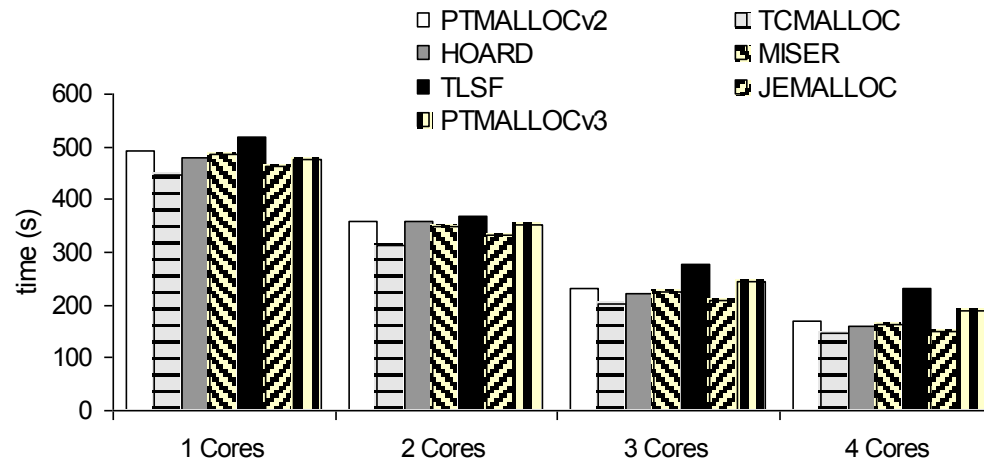
- These results show that memory request sizes in the three applications are predominantly smaller than 64 bytes.
 - specially, the requests for **twenty-four bytes** are the most prevalent observed in the three applications.
- Due to the high number of allocations per second (App1=4600, App2=80000, App3=50000), the charts just show the dataset related to **one-minute load**.
- **The observed patterns indicate that good allocators for these applications should be optimized for smaller memory blocks.**

Performance Tests

- We process 20,000 stock orders and measured:
 - the time spent by the middleware
 - memory consumption per application
 - number of memory fragmentation events per application
- We did these tests for each different number of cores
 - 1 up to 4.

Performance Tests (cont'd)

Middleware Throughput



- TCMalloc shows the best performance for all number of cores, followed by Jemalloc and Hoard.
- Hoard improves as the number of cores increases over two, most probably because it starts having a higher number of local heaps.
- These three allocators implement local heaps, which are responsible for serving requests of small size in a faster way, thus being appropriate to the middleware memory usage pattern.
- For all number of cores, the Ptmallocv2 shows worse performance than TCMalloc and Jemalloc.

Performance Tests (cont'd)

- We used the ANOVA and Tukey tests to assess the statistical significance of the results.
- Note that to satisfy the ANOVA assumptions we applied the $1/x$ transform on the data sample.
- We considered statistically significant $p\text{-value} < 0.05$ for both ANOVA and Tukey tests.

Performance Tests (cont'd)

ANOVA of the Execution Time for the Allocators

Source	DF	SS	MS	<i>F</i>	p-value
allocator	6	0.00005745	0.00000958	1238.68	<.0001
# of cores	3	0.00083792	0.00027931	36130.3	<.0001
allocator vs. # of cores	18	0.00004319	0.00000240	310.37	<.0001
Error	392	0.00000303	0.00000001		
Total	419	0.00094159			

Performance Tests (cont'd)

Tukey Test Results of Allocators and Number of Cores to Compare Execution Time

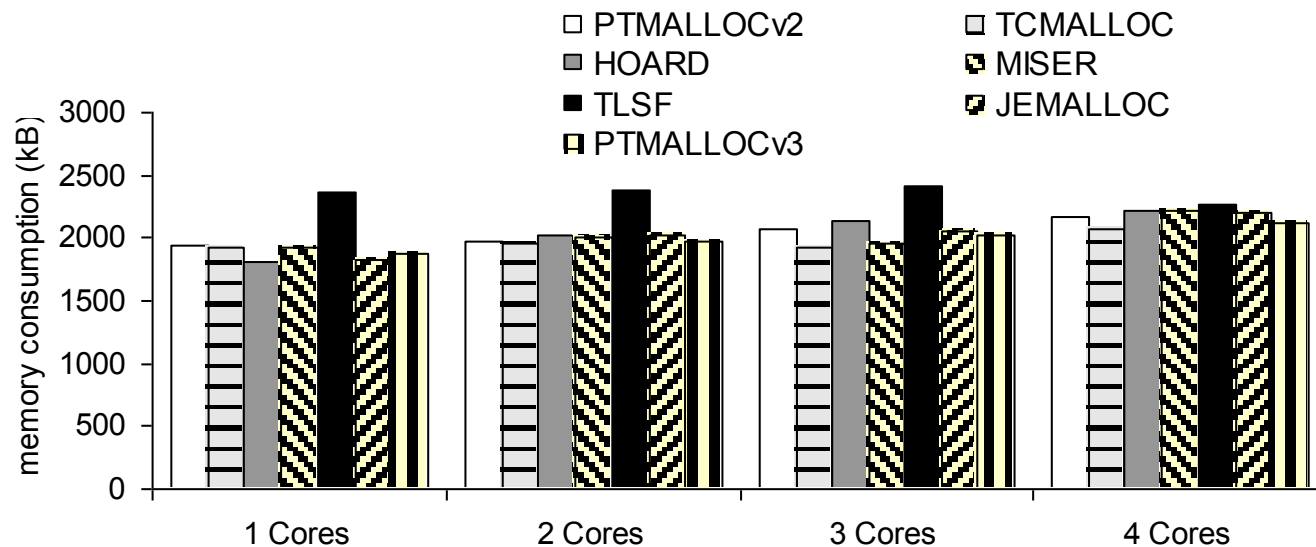
Allocator / Core	1	2	3	4
Hoard	479.63 (Ac)	357.53 (Bab)	220.33 (Ce)	159.33 (Dd)
Jemalloc	461.47 (Acd)	332.19 (Bc)	210.30 (Cf)	151.43 (De)
Ptmalloc2	492.94 (Aab)	358.52 (Bab)	233.02 (Cc)	201.81 (Db)
Ptmalloc3	474.94 (Abc)	350.71 (Bb)	243.69 (Cb)	190.07 (Dc)
Miser	486.72 (Abc)	348.52 (Bbc)	226.33 (Cd)	162.07 (Dd)
TCMalloc	445.42 (Ad)	311.56 (Bd)	201.81 (Cg)	146.30 (Df)
TLSF	519.50 (Aa)	368.76 (Ba)	276.42 (Ca)	232.40 (Da)

Lower and upper case letters compare values in the same column and line, respectively. If two or more allocators share at least one letter, then there is no statistical difference (p-value > 0.05) between them; otherwise they are statistically different (p-value < 0.05).

- TCMalloc, Jemalloc, Hoard are considered statically different for any core (p-value < 0.05).

Performance (cont'd)

Middleware Memory Consumption



- Except for one core, in all other tests the TCMalloc shows the lowest average memory consumption, and the TLSF the highest one.

Performance (cont'd)

ANOVA of Memory Consumption for the Allocators

Source	DF	SS	MS	<i>F</i>	p-value
Allocator	6	0.08418	0.01403	340.09	<.0001
# of cores	3	0.04416	0.01472	356.78	<.0001
allocator vs. # of cores	18	0.02605	0.00145	35.08	<.0001
Error	112	0.00462	0.00004		
Total	139	0.15901			

Performance (cont'd)

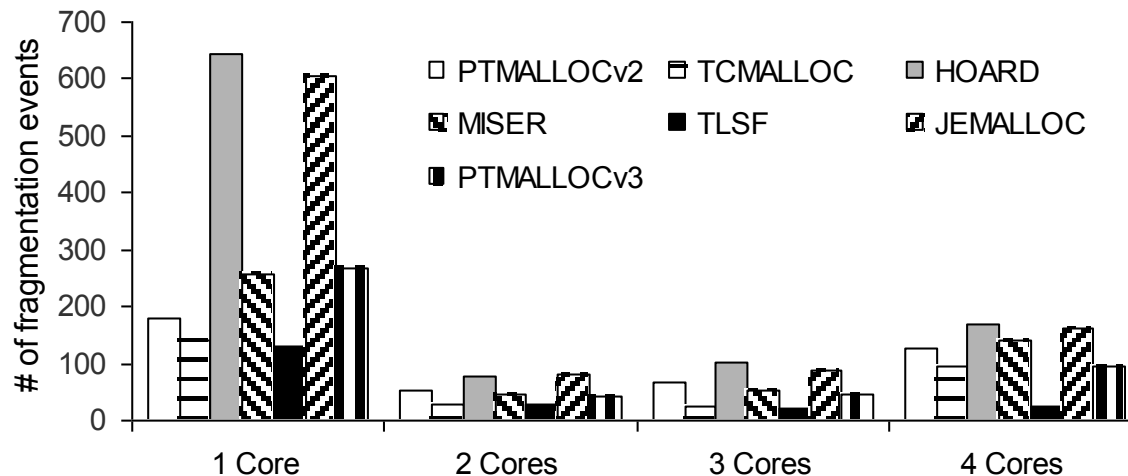
Tukey Test Results of Allocators and Number of Cores to Compare Memory Consumption

Allocator / Core	1	2	3	4
Hoard	1809.0 (Cd)	2028.0 (Bb)	2139.8 (Ab)	2212.2 (Aab)
Jemalloc	1820.2 (Dd)	2028.6 (Cb)	2049.8 (Bc)	2200.6 (Aab)
Ptmalloc2	1940.0 (Cb)	1979.4 (Cb)	2070.0 (Bbc)	2164.2 (Abc)
Ptmalloc3	1869.6 (Ccd)	1965.6 (Bb)	2025.0 (Bc)	2112.0 (Acd)
Miser	1922.4 (Bbc)	1999.0 (Bb)	1957.2 (Bd)	2212.6 (Aab)
TCMalloc	1917.8 (Bbc)	1963.4 (Bb)	1927.0 (Bd)	2072.6 (Ad)
TLSF	2362.2 (Ba)	2387.4 (Ba)	2409.8 (Ba)	2272.0 (Aa)

- TCMalloc shows no statistical difference when running on one, two or three cores. The same behavior can be observed with Miser and TLSF.
- **Note that Ptmallocv3 is the allocator with the topmost best values for all numbers of core.**

Performance Tests (cont'd)

Memory Fragmentation Level



- TLSF shows the lowest level of memory fragmentation, followed by TCMalloc, possibly because it uses only one heap from where all requests are served, simplifying its address space.
- Hoard and Jemalloc show the worst performance for all number of cores. This result is consistent considering that Jemalloc is strongly based on the Hoard design.

Performance Tests (cont'd)

ANOVA of Memory Fragmentation for the Allocators

Source	DF	SS	MS	<i>F</i>	p-value
Allocator	6	0.1218	0.0203	158.52	<.0001
# of cores	3	0.1688	0.0563	439.28	<.0001
allocator vs. # of cores	18	0.0358	0.0020	15.55	<.0001
Error	112	0.0143	0.0001		
Total	139	0.3408			

Performance Tests (cont'd)

Tukey Test Results of Allocators and Number of Cores to Compare Memory Fragmentation

Allocator / Core	1	2	3	4
Hoard	645.0 (Aa)	78.2 (Cab)	103.0 (BCa)	167.8 (Ba)
Jemalloc	604.6 (Aab)	81.8 (Da)	86.4 (Ca)	160.6 (Ba)
Ptmalloc2	179.8 (Abc)	52.6 (Bbc)	67.0 (Bab)	128.2 (Aa)
Ptmalloc3	267.4 (Aabc)	43.8 (C _c)	44.6 (C _c)	95.2 (B _a)
Miser	255.4 (Aabc)	46.0 (B _c)	53.2 (Bb _c)	142.2 (A _a)
TCMalloc	140.2 (Ac)	27.6 (Bd)	23.2 (Bd)	96.4 (Aa)
TLSF	130.4 (Ac)	29.2 (Bd)	21.2 (Bd)	25.8 (Bb)

- TLSF executions for 2, 3, and 4 processors are not statistically different.

Conclusion

- **TCMalloc showed the best results in all evaluated criteria.**
- **Jemalloc and Hoard show very good performance in terms of response time, but they present high memory consumption and fragmentation.**
 - in long lasting applications fragmentation should be the smallest possible, because it contributes significantly to the memory exhaustion in long-term executions.
- **Hence, we consider the Ptmallocv3 as a second option among all the evaluated allocators.**

Final Remarks

- We present a systematic approach to evaluate UMA, highlighting the importance of the characterization phase.
- We compare the allocators based on their response time, memory consumption, memory fragmentation, and a combination of these aspects on different numbers of processor cores.
- This four-dimension approach allows the experimenter to have a better view of each allocator benefits and limitations, per application.
- Finally, the results obtained in the stock trading middleware case study are discussed in general terms (e.g., request sizes and number of requests)
 - It allows one to apply these results to applications showing similar allocation patterns.
 - e.g., Jemalloc should not be used for a multithreading application running in a single core machine, whose request sizes are less than 64 bytes and where fragmentation is a major concern.

Thank You!

Rivalino Matias Jr.

rivalino@fc.ufu.br