Memory allocation: a bottleneck for multithreading.

Written by **SID-LAKHDAR Riyane Yacine** (M2 MoSIG: ENSIMAG / UJF)

Directed by **KRAKOWIAK Sacha** (LIG, team ERODS)

Objectives

The aim of this study is to analyze and to compare the efficiency of different memory allocation strategies. Our objective is to compare their time and memory performance for a specific lock implementation.

Context

- **Complex multi-core processors:** challenge to maintain and localize data despite of the large number of separated memory component.

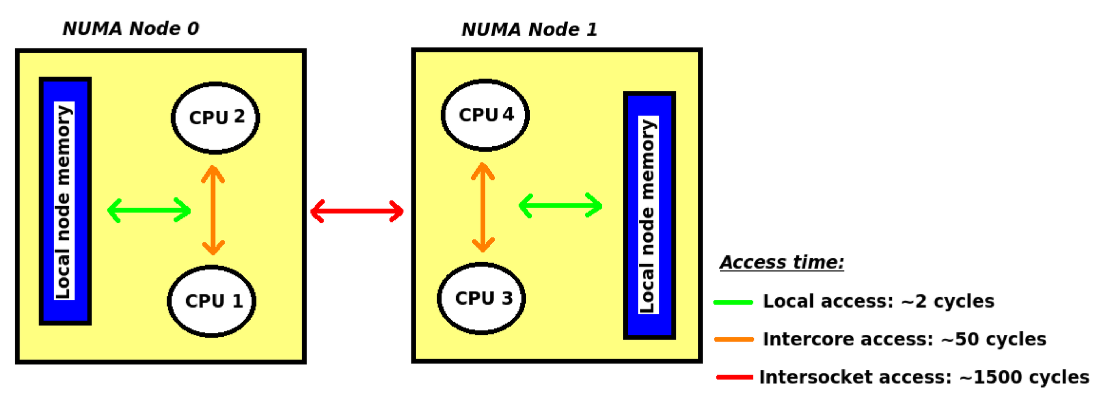

Figure 1: Simplified NUMA model (only $\frac{3}{8}$ memory access types)

- **Complex concurrent data structure:** Accessed simultaneously by many threads ⇒ the sequential memory allocator becomes a bottleneck

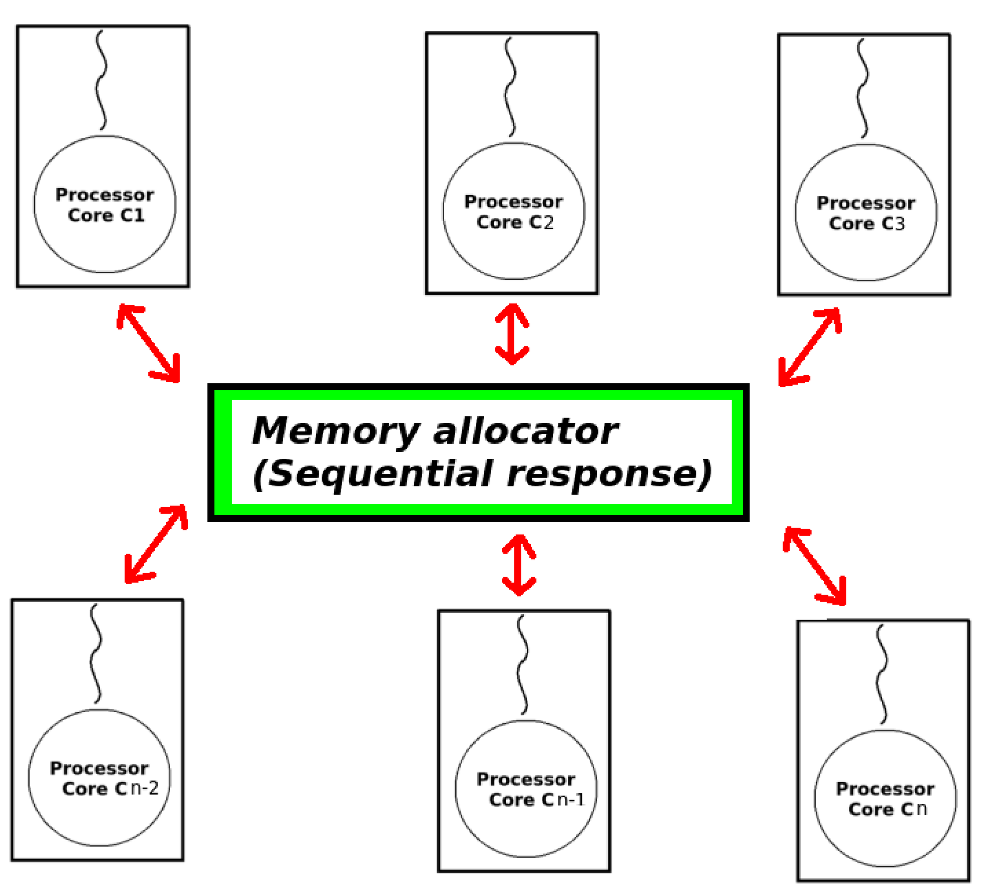

Figure 2: Sequential memory allocator: bottleneck for concurrent thread

- **Non-uniform memory access (NUMA):** Non uniform access time may severely harm the performance of highly optimized thread synchronization algorithms.

Hardware issues

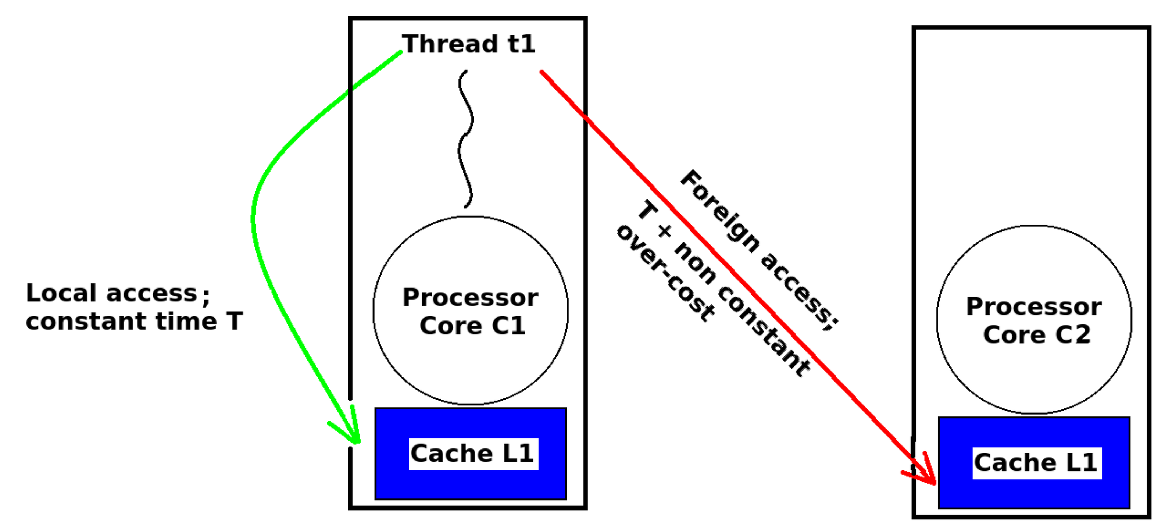

Figure 3: Core locality

- **Cache and memory coherence:** over-cost due to synchronization of separated hardware elements
- **Foreign cache access:** client-server request model (or Remote Memory References).

Global architecture

- Improve thread locality (1 buffer per core)
- Decrease synchronization for allocation and memory access (Treiber stack designed to need no synchronization between any core)

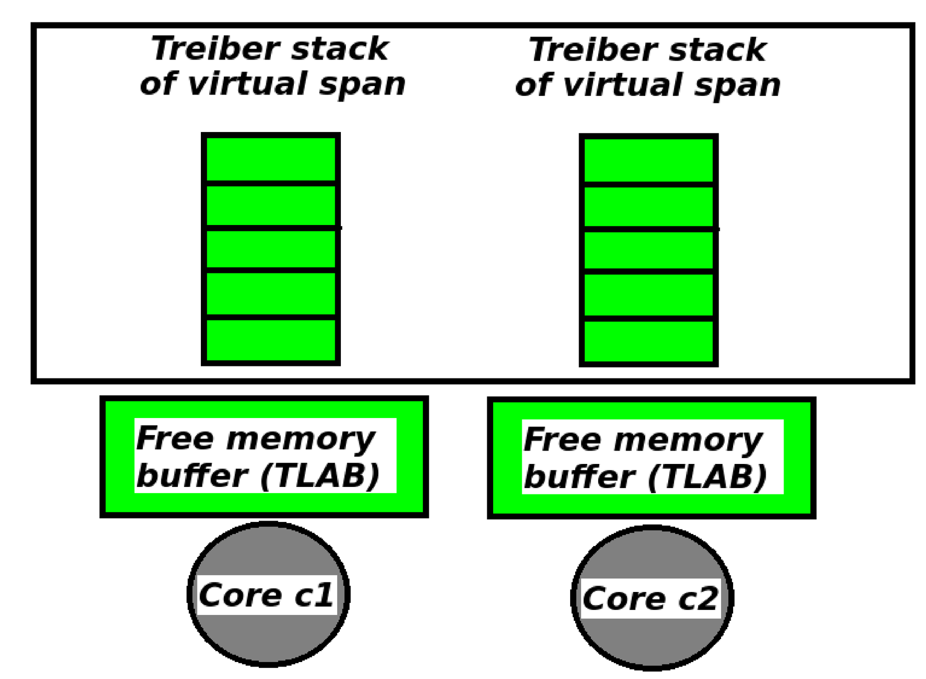

Figure 4: Free memory pool: shared by all the cores

Experimental results

- Average time improvement of 20 → 40 % compared to the c standard library
- Average time improvement of 10 → 20 % compared to existing implementation designed for multithreading
- Very different results depending on the environment

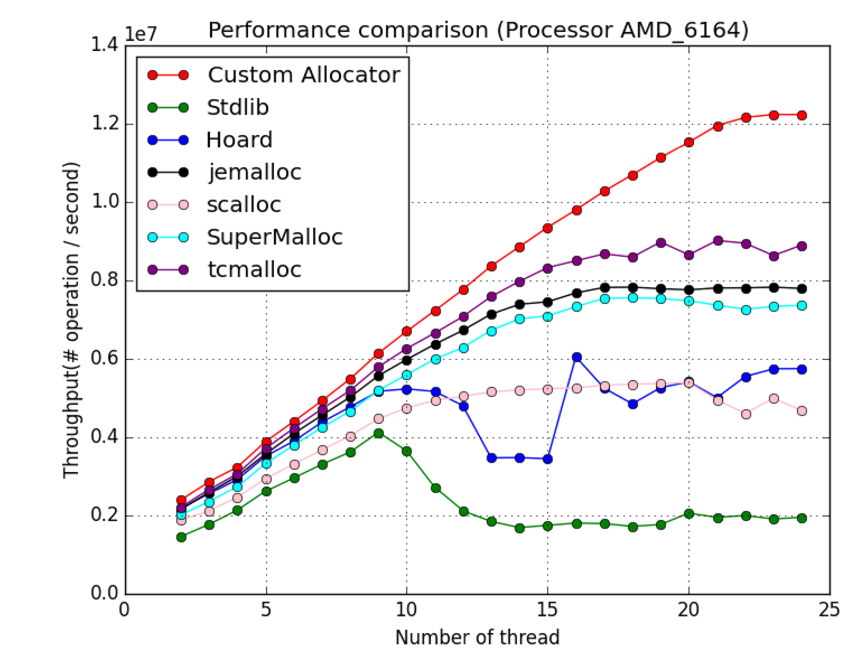

Figure 7: Average throughput comparison between memory allocators

Designed test environment

Interest of the environment

- The memory allocation is a complex process that involves many other components of the system (hardware and software).
- Difficult to test the allocator without being influenced by the performances of the other parts

Formal definition

- No more threads than cores:
- Each thread always run on the same core

Software issues

- **Contention and synchronization over-cost:** The allocator is shared between all threads.
- **Context switch cost:** depends on the processor architecture (task state segment) and the data transfer throughput (memory word size).
- **Fragmentation and memory blowup**
- **False sharing:** Occurs when multiple processors share a word in the same cache line without actually sharing data (useless synchronization over-cost).

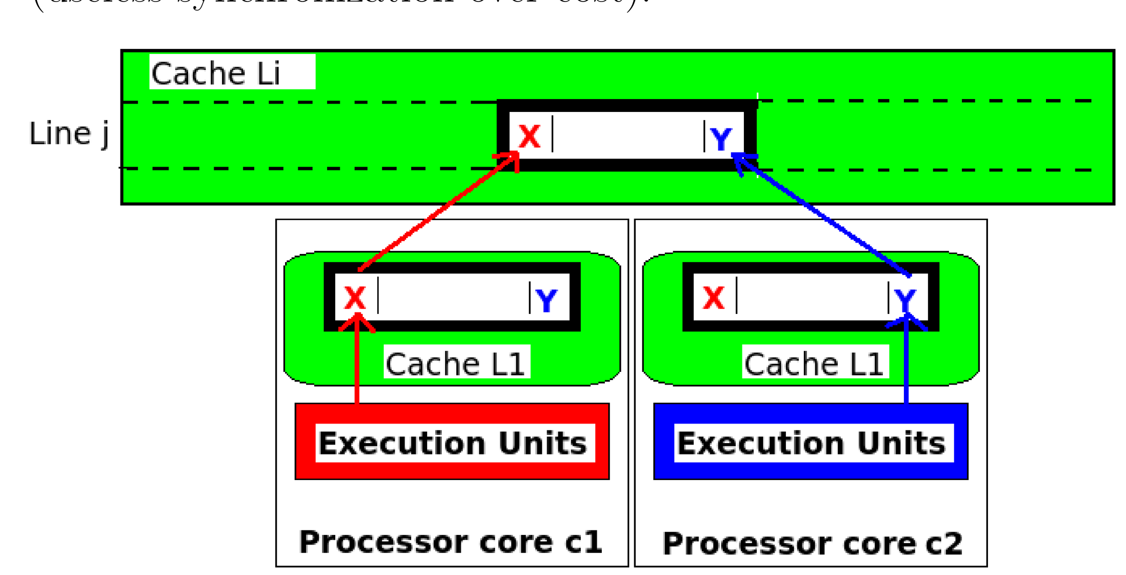

Figure 5: False sharing representation

Core local Treiber stack

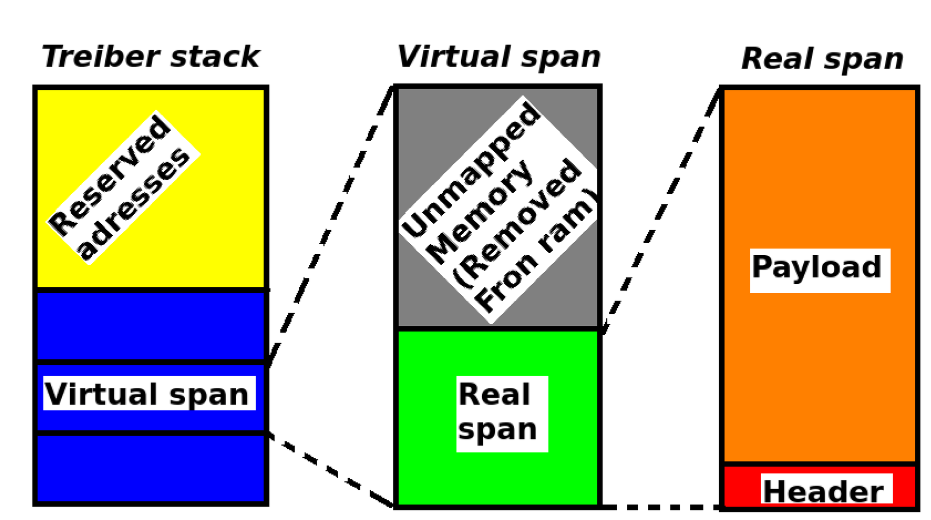

Figure 6: Structure and content of a Treiber stack

- Delegate memory allocation to user program ⇒ No system call
- Constant size span ⇒ no false sharing + no external fragmentation
- Treiber stack per core ⇒ constant time allocation

Conclusion

During this study, we have

- Designed and explored memory allocation strategies
- Highlighted the existing designs that limit the performances of multithreaded applications

We have also implemented an allocator with

- Interesting time and memory gain compared to the existing ones
- Stable performance (interesting for research purposes)

References

[1] D. Wood D. Sorin, M. Hill. *A Primer on Memory Consistency and Cache Coherence*, volume 1. Morgan & Claypool, 3 edition, 2011.

[2] L. Yerushalmi D. Hendler, N. Shavit. *A Scalable Lock-free Stack Algorithm.* PhD thesis, Sun Microsystems (USA), 2014.

[3] A. Macedo L. Araujo T. Ferreira, R. Matias. *An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications.* PhD thesis, Federal University of Uberlandia Uberlandia, (Brazil), 2014.

[4] A. Schiper D. Petrovic, T. Ropars. *On the Performance of Delegation over Cache-Coherent Shared Memory.* PhD thesis, Ecole Polytechnique Federale de Lausanne (Switzerland), 2014.