DOI:10.1145/3636500

To view the accompanying paper, visit doi.acm.org/10.1145/3611018 rh

Technical Perspective
Learning-Based Memory Allocation for C++ Server Workloads

By Doug Lea

MEMORY MANAGEMENT SPANS the layers of computing platforms, ranging across hardware components that map logical addresses to physical memory locations, operating system components that track and control regions (usually in the form of "pages"), and run-time systems and languages that provide simple APIs and/or language constructs to process higher-level objects in terms of lower-level memory segments. An allocation ("new" or "malloc") is performed by placement—choosing a memory address heading a contiguous segment of (at least) a given size. A deallocation ("delete" or "free," perhaps initiated by garbage collectors or other mechanisms that detect unused objects) triggers bookkeeping to enable future memory reuse. Some languages and systems additionally support moving (copying) previously allocated objects to new locations, and automatically adjusting pointers to them accordingly.

The primary objective of memory management is fitting all requested memory segments within a given space, or nearly equivalently, minimizing unusable gaps surrounding allocated objects. When gaps accumulate, there may be enough aggregate memory to satisfy a new allocation request, but not enough contiguously mappable space. Performance is mainly a function of time/space overhead per allocation and deallocation. Even in the simplest cases, management is equivalent to NP-complete bin-packing problems for which there is rarely a computationally feasible optimal solution. All allocator designs reflect trade-offs between quality (footprint) and performance, along with other policy and system constraints. Evaluations are intrinsically empirical, based on measurements of actual programs.

One strategy for reducing overhead while maintaining a compact footprint is to use bulk operations that perform allocation and/or deallocation steps for possibly many objects all at once. This is seen in stack allocation in most languages, where local variables are contiguously placed (normally according to compile-time layout rules) in a pushed stack frame and then deallocated all at once by popping the frame. This technique has become so entrenched in languages and systems that it is usually supported by different programming constructions than those for other "heap" objects and has been extended in some languages to include extended scoping rules and/or shadow stacks, applicable when all memory accesses meet generalized rules for block structuring.

A different form of bulk processing is available using virtual memory hardware and operating system functionality. Systems support mappings to physical memory in increasingly large units per operation, although often restricted to a few choices of sizes, as in Linux 2MB "huge" pages. Creating and managing the right number of regions (pages) of the right size and contents can lead to very efficient memory management while still maintaining an acceptable total footprint.

If the lifetime of every memory segment were known at allocation time, then this would not be difficult to carry out: Place objects with (nearly) coextensive lifetimes in the same mapped region, and unmap them at the same time when they are all freed. But lifetime information for arbitrary heap allocation requests is not generally available, or even knowable by compilers. On the other hand, using even imperfect information has potential value in the design of memory managers that outperform those that do not employ lifetime considerations during allocation.

The following paper is full of good ideas, both large and small, about neural network design and deployment.

The following paper is the first to explore this idea in depth, as implemented and evaluated in the "Llama" allocator. The authors develop a neural net-based statistical prediction scheme for lifetimes based on selected properties of calling contexts and use it to control placement and subsequent management. The main challenges are to achieve sufficient accuracy to reduce footprint versus alternatives, while ensuring the overhead for learning and acting upon placement rules does not outweigh benefits. The empirical evaluation of Llama clearly shows benefits at least in the context of large C++ server applications, for which memory management quality and performance are serious practical concerns.

The paper is full of good ideas, both large and small, about neural network design and deployment, tolerating mispredictions using lifetime classes with adjustable deadlines, and incorporating these ideas in a high-quality memory manager that preserves fast-path performance in common cases. The authors also set the stage for a variety of potential improvements, as well as applications to other forms of memory management. As one example, the overall structure of Llama is surprisingly reminiscent of those of "generational" garbage collectors that copy straggling objects to new regions rather than delay unmapping of lifetime classes. But it improves upon this approach by initially placing objects in regions in ways that can greatly reduce stragglers, which might lead to advances in these forms of memory managers as well. C

Doug Lea is a professor and chair of the Department of Computer Science at the State University of New York, Oswego, NY, USA.