COMPSCI 590S Systems for Data Science **Fall 2016**

Lecture 1: Memory Allocation

Lecturer: Emery Berger *Scribe(s): Bruno Silva,Jim Partan*

1.1 Review of Custom Allocators

We reviewed per-class allocators, custom pattern allocators, and region allocators from last lecture.

1.2 DieHard Allocator

Allocators can also be used to avoid problems with unsafe languages. C and C++ are pervasive, with huge amounts of existing code. They are also memory-unsafe languages, in that they allow many errors and security vulnerabilities. Some examples include double `free()`, invalid `free()`, uninitialized reads, dangling pointers, and buffer overflows in both stack and heap buffers.

DieHard is an allocator developed at UMass which provides (or at least improves) soundness for erroneous programs.

There are several hardware trends which are occurring: multicore processors are becoming the norm, physical memory is relatively inexpensive, and 64-bit architectures are increasingly common, with huge virtual address spaces. Meanwhile, most programs have trouble making full use of multiple processors. The net result is that there may soon be unused processing power and enormous virtual address spaces.

If you had an infinite address space, you wouldn't have to worry about freeing objects. That would mostly eliminate the double `free()`, invalid `free()`, and dangling pointer bugs. And if your heap objects were infinitely far apart in memory, you wouldn't need to worry about buffer overflows in heap objects.

DieHard tries to provide something along these lines, within the constraints of finite physical memory. It uses randomized heap allocation, so objects are not necessarily contiguous in virtual memory. Since the address space is actually finite, objects won't actually be infinitely far apart, and buffer overruns might actually cause collisions between heap objects. But this is where the multicore processors come in: With the unused processor cores, run multiple copies of the application, say three copies, each allocating into their own randomized heap. So the heap errors are independent among the three copies of the application. All copies get the same input, and the output is the result of voting among the three copies of the program. If one instance of the application disagrees with the other two, it is killed, since there was likely a collision between heap objects in that one. Similarly, if one instance dies with a segfault or other error, the others remain running. Surviving copies can be forked to replace copies which were killed off, though this reduces the independence among copies.

This is transparent to correct applications, and allows erroneous applications to survive longer, which will make their users happier. Furthermore, when a copy dies or is killed off, the error can be noted and sent to the software maintainers automatically.

DieHard increases the chances of turning problematic errors into benign errors. It trades memory space for robustness, which is probably fine as memory prices continue to drop. It uses randomization and replication to provide fault-tolerance.