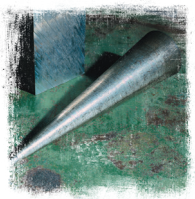


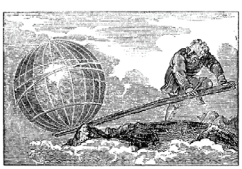
Persistent Memory Allocation

LEVERAGE TO MOVE A WORLD OF SOFTWARE

TERENCE KELLY WITH SPECIAL GUEST BORERS
ZI FAN TAN, JIANAN LI, AND HARIS VOLOS

"Give me a place to stand, and I will move the earth"

— Archimedes



LAYERS AND LEVERS

This episode of Drill Bits presents a new persistent memory allocator that enables a new capability for scripting languages. The new allocator is suitable for serious use, yet it is more concise and simpler than most alternatives. The new capability is transparent on-demand persistence for interpreted scripts with zero effort from script authors. Persistent memory allocation and persistent scripting demonstrate, at two very different software layers, that the right interfaces multiply programmer impact by leveraging mountains of existing software and hardware. Both the new persistent memory allocator and a persistent script interpreter that uses it are available as open-source software.

SCRIPT MEMORABILIA

Scripting allows you to write clear, concise, and correct code quickly and conveniently. Scripting languages enhance productivity via keen instincts for programmer intentions. For example, they create and initialize variables as needed without explicit declarations. *Persistent scripting* brings similar do-what-I-mean convenience to data used across multiple script executions.

The AWK program in figure 1 captures the essence of classic scripting chores such as log file processing. Its input block reads strings, one per input line, and uses associative

FIGURE 1: AWK SCRIPT WITH TYPICAL LOG-PROCESSING ELEMENTS

```
{                       # Input block:  Executes once per input line ($0).
  if (! ($0 in id))     # Assigns serial numbers to unique strings and
    id[$0] = ++n;       # counts string frequencies using associative
  freq[$0]++;           # arrays id[] and freq[] and counter variable n.
}
END {                             # END block:  Executes after all input
  print n;                        # has been read.  Prints summary report
  for (s in id)                   # of every unique string seen, its
    print id[s], s, freq[s];      # serial number, and its frequency.
}
```

arrays to assign serial numbers to unique strings and to count their frequencies. After all input has been processed, the `END` block prints a summary report.

Efficiency demands persistence in common scenarios where inputs arrive periodically and summaries are cumulative—say, a daily report must cover all past logs and a new log arrives every day. Naïvely processing all logs every day would be inefficient; an incremental approach is needed. For the script of figure 1, arrays `id[]` and `freq[]` and counter `n` must somehow propagate from yesterday's execution to today's run.

The persistence facilities in popular scripting languages flout the casual spirit of scripting because they demand explicit fuss with individual variables. For example, Python and Perl can bind arrays to `dbm` databases, but coding explicit persistence and wrangling per-array databases are a bother. Why can't a script simply remember programmer-defined variables from one execution to the next?

Persistent scripting provides the right interface for scripts to remember variables across executions. A new command-line option informs the interpreter that

script-defined variables reside in a persistent heap. When a script runs with the new interpreter flag, the script begins execution pre-populated with variables from the persistent heap; when the script terminates, the persistent heap retains the state of its variables for its next execution. Scripts remain oblivious to persistence, so all existing scripts obtain the benefits of persistence without modification. Persistent heaps are separate from scripts and may be shared freely among unrelated scripts. That's exactly what is needed for scripts like that shown in figure 1.

Despite its attractions, persistent scripting would be impractical if it were too difficult to implement or couldn't be widely deployed. Does the new command-line option require extensive modifications to interpreters? Do persistent heaps require rare, unconventional NVM (non-volatile memory)?

Fortunately not. A persistent memory allocator *with the right interface* makes it remarkably easy to retrofit persistent scripting onto a widely used, feature-rich, production-grade interpreter. Furthermore, such an allocator need not rely on exotic hardware.

We have implemented persistent scripting in the GNU `AWK` interpreter, `gawk`. Our "persistent memory `gawk`" (`pm-gawk`) affects roughly 70 lines in a source code base of 91,000 LOC. A companion paper on `pm-gawk` presents performance evaluations of both conventional hardware and Optane NVM and details the benefits of persistent scripting.[12] We are working with the `gawk` maintainer to integrate `pm-gawk` into the official distribution. This

column describes the new persistent memory allocator that makes `pm-gawk` possible.

LOW-LEVEL LEVERAGE

Persistent memory enables in-memory data structures to outlive the processes that access them. Persistent memory programming simplifies applications by obviating the need for different formats and programming concepts to govern persistent data.

Our persistent memory allocator, `pma`, presents a familiar interface compatible with existing C/C++ code: It simply provides `pma_*` replacements for the standard `malloc`, `calloc`, `realloc`, and `free` functions, plus an initialization routine and `get`/`set` functions to access a *root pointer* whereby applications locate data on `pma`'s persistent heap.

Figure 2 illustrates the simplicity and familiarity of `pma` with a complete C program that maintains a persistent linked list. List nodes hold string keys and numeric values (line 1). Function `find` traverses the list using pointers-to-pointers to list nodes (lines 3– 6). User commands, parsed on lines 27–33, trigger functions that set, print, and delete key/value pairs (lines 8–20). The program is nearly identical to a conventional program that maintains the same list in ephemeral memory. For example, `pma_malloc` and `pma_free` are used exactly like their standard counterparts (lines 10 and 20).

Persistence requires a small amount of code to initialize `pma`'s persistent heap and access the heap's root pointer. Initialization function `pma_init` (line 23) must be called before any other `pma` function. It specifies a file, `heap.pma`,

FIGURE 2: PERSISTENT LINKED LIST PROGRAM (#INCLUDES OMITTED)

```
typedef struct ln { struct ln *next; long val; char key[]; } ln_t;

static ln_t **find(ln_t **h, char *k) {  // On return **h is sought
  while (*h && strcmp((*h)->key, k))     // node, or *h == NULL if
    h = &((*h)->next);                   // not found.
  return h; }

static void set(ln_t **h, char *k, long v) {
  ln_t **p = find(h, k);  // insert new node if key not found
  if (!*p) { *p = (ln_t *)pma_malloc(sizeof **p + 1+strlen(k));
             assert(*p);  (*p)->next = NULL;  strcpy((*p)->key, k); }
  (*p)->val = v; }

static void print(ln_t **h, char *k) {
  ln_t **p = find(h, k);
  if (*p) printf("\"%s\" -> %ld\n", k, (*p)->val); }

static void del(ln_t **h, char *k) {
  ln_t *t, **p = find(h, k);
  if (*p) { t = *p;  *p = t->next;  pma_free(t); } }

int main(int argc, char *argv[]) {
  ln_t *L;  int i, r = pma_init(2, "heap.pma");
  if (r) { fprintf(stderr, "pma init->%d errno=%d\n", r, pma_errno);
           return 1; }
  L = (ln_t *)pma_get_root();  // list entry point; NULL for new heap
  for (i = 1; i < argc; i++) {
    char *a = argv[i], k[9];  long v;  // arg, key, value
    if      (2 == sscanf(a, "s %8s %ld", k, &v)) set  (&L, k, v);
    else if (1 == sscanf(a, "p %8s",     k    )) print(&L, k   );
    else if (1 == sscanf(a, "d %8s",     k    )) del  (&L, k   );
    else    fprintf(stderr, "bad argument \"%s\"\n", a);
  }
  pma_set_root(L);
  return 0;
}
```

to contain `pma`'s persistent heap. A heap is born as a sparse file whose size is a multiple of the system page size; create one with, for example, "`truncate -s 409600 heap.pma`" on the command line.

The least familiar aspect of `pma`'s persistent heap is the root pointer, accessed via `pma_get_root` and `pma_set_root`. Applications must ensure that all "live" (in-use) persistent memory is reachable via the root. The

program of figure 2 maintains a linked list on the persistent heap, so every time it executes it must obtain the list entry point from `pma_get_root` (line 26). Before terminating, the program must update the root (line 34).

The root pointer isn't unique to `pma`; most persistent heap designs, including Intel's PMDK (Persistent Memory Development Kit),[11] have a similar feature. Fortunately the root isn't as tricky as newcomers often fear, even when retrofitting persistence onto complex software designed for ephemeral memory. Transforming `gawk` into `pm-gawk`, for example, was remarkably easy because `gawk` already had a single entry point into its symbol table of script variables; it was easy to equate this entry point with `pma`'s root pointer. The case of `pm-gawk` resembles many similar experiences with a wide range of existing software: Retrofitting persistence is often quite easy if you use the right allocator. For example, sliding a malloc-compatible persistent heap such as `pma` beneath C++ STL containers, thereby creating persistent containers, takes a few lines of straightforward code.[5]

The most valuable aspect of `pma`'s interface is also the most mundane: compatibility with standard `malloc`. Many persistent memory allocators return *offsets* relative to the heap's base address, whereas `malloc` returns *absolute-address* pointers. The main advantage of "offsettish" persistent heaps is relocatability: They can be placed anywhere in virtual memory. Their main disadvantage, of course, is incompatibility with the enormous installed base of pointer-based applications and libraries. "Pointerish" allocators such as `pma` make the opposite tradeoff: An initialized `pma` heap must always

be mapped at the same memory address, and pma plays nicely with conventional pointer-based software.

Another attraction of pma, and another contrast with many alternatives, is that pma supports persistent memory programming on conventional hardware.[5] Allocators that exclusively target non-volatile memory can strive for improved performance by exploiting special features of NVM hardware. The downside of such specialization, of course, is incompatibility with the enormous installed base of conventional computers. A hardware-agnostic allocator such as pma is better suited for software, such as pm-gawk, that must run on the widest possible range of machines.

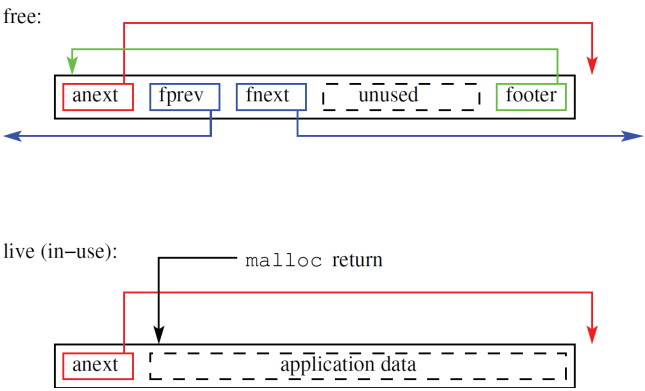
UNDER THE HOOD

Given an uninitialized heap file, pma_init first finds a large unused gap in the caller's address space and maps the heap into the middle of it. Then pma_init initializes the file by dividing it into a header and an allocatable region. The header contains the virtual address at which the heap must always be mapped, the root pointer, and allocator metadata including free-list heads. Allocatable blocks of memory are grouped into size classes, each with its own free list. The initial allocatable region—the entire heap file beyond the header—is a single block on the free list of the appropriate size class. Following tradition, we refer to this never-before-allocated memory as the "wilderness" block.[14]

Allocation functions such as pma_malloc scan free lists until an adequately large block is found. If this block is larger than required to fulfill the allocation request, it is split into a block returned to the caller and a remainder that is replaced onto the appropriate free list. The sizes of

3

FIGURE 3: FREE AND LIVE BLOCKS (HEADER BIT FLAGS NOT SHOWN)



all blocks are multiples of an eight-byte machine word.

Figure 3 shows the layout of free and live blocks. Both have a header field `anext` (red), which points just beyond the block. All blocks, both live and free, reside on an address-ordered singly linked list defined by these headers. Unused low-order bits of `anext` contain flags indicating whether the block is live or free, and whether the previous block on the address-ordered list is live or free. Free blocks have three additional pointer fields: `fprev` and `fnext` (blue) link the block into a free list, and a footer (green) points back to the block's first byte.

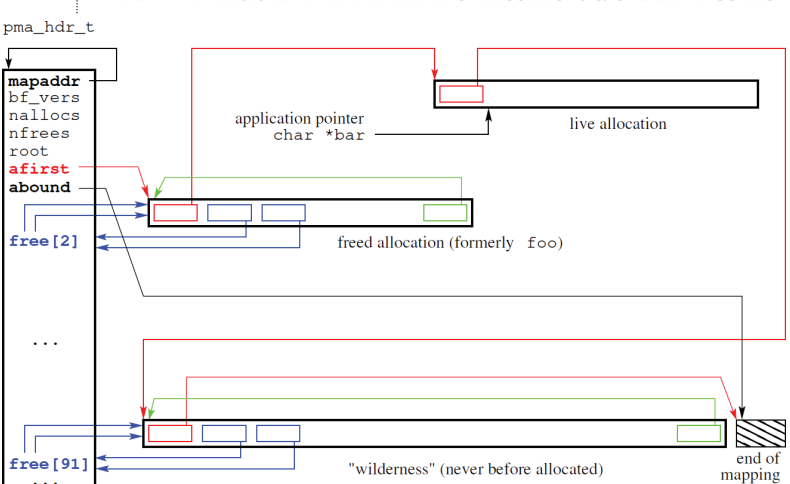
Block footers, headers, and the bit flags in the headers together allow deallocated blocks to be immediately coalesced (merged) with free adjacent blocks when `pma_free` is called. Coalescing freed blocks reduces the need to encroach on the wilderness, which reduces the number of pages in the initially sparse heap file that must

be backed by physical storage. Coalescing brings a further benefit: Any sequence of allocations and deallocations that frees all allocated blocks will return the persistent heap metadata to its initial state. This "reversibility" property greatly facilitates debugging of both `pma` itself and the applications that use it. Most memory allocators do *not* have this property, often because they treat small and large blocks as special cases. Together, coalescing and uniform treatment of block sizes yield reversibility gratis.

Figure 4 depicts the state of `pma`'s persistent heap after a 100-page (409,600-byte) heap file is initialized and the following sequence of calls is made:

```
char *foo, *bar;
foo = (char *)pma_malloc(33);
bar = (char *)pma_malloc(33);
pma_free(foo);
```

FIGURE 4: **PERSISTENT HEAP AFTER TWO ALLOCATIONS & ONE DE-ALLOCATION**



The foo and bar requests are satisfied by biting six-word blocks from the left-hand side of the wilderness: five words to satisfy the rounded-up 33-byte request and one word for the red header field. Traversing the red address-ordered list, starting from the list head (afirst) on the left, visits these blocks and the remaining wilderness. Because the foo block has been freed, it has a footer and resides on the free list appropriate for its size. The bar block remains in use. If freed, it will be coalesced immediately with the freed former foo block that precedes it on the address-ordered list and with the remaining wilderness that follows it, returning the persistent heap's metadata to its initial state.

It will be easy to determine the candidates for coalescing if pma_free(bar) is called, because the already-free block, the bar block, and the wilderness block are adjacent to each other in the address space. Decrementing the bar pointer by one machine word leads to the block's anext header, which points to the right-hand candidate, the wilderness. The low bits of the bar block's header indicate that the previous block is free, so moving one word to the left of bar's header takes you to the former foo block's footer, which points to its header.

TRADEOFFS AND EXTENSIONS

Beyond the "pointerish" vs. "offsettish" interface choice already discussed, pma makes several further tradeoffs. Most importantly, the current implementation favors simplicity over sophistication. Compared with most other allocators, pma's design is more straightforward and its code more concise. An elaborate implementation would obscure the fundamental simplicity of persistent memory

allocation. Adding sophistication as experience demands—for example, special-case treatment for small blocks to reduce internal fragmentation—is easier than removing needless complexity present at inception.

Like `dlmalloc`,[10] the foundation of today's widely used `glibc` allocator,[2] `pma` is serial; multithreaded applications can protect `pma_*` calls with a mutex. Beware, however, of subtle traps where parallelism meets persistence. In particular, conventional synchronization primitives aren't designed to be embedded in persistent `struct`s; ordinary mutexes may depend on non-persistent memory allocation and may require re-initialization or unlocking every time a persistent heap is mapped into memory. Therefore, fine-grained locking strategies that, for example, embed mutexes in linked data structures accessed via hand-over-hand locking[6] are no longer straightforward in persistent memory. Fortunately, fine-grained locking isn't necessary for every application.[1] Persistent scripting requires no locking whatsoever in single-threaded scripting languages such as `gawk`.

Crash tolerance, if required, is handled outside `pma` in whatever manner best suits particular applications. For scripting, both ordinary and persistent, it suffices to back up important files such as persistent heaps before/after scripts modify them. Applications that must checkpoint a persistent heap *during* execution may borrow the production-strength crashproofing mechanism used in the `gdbm` database.[7] Test whatever mechanism you choose against the failures it purports to tolerate. Injecting process crashes and OS kernel panics is easy enough using "kill -9" and `/proc/` interfaces. A cheap yet robust

hardware platform makes it easy to automate testing against sudden whole-system power failures.[8]

DRILLING DEEPER

Learn persistent memory programming by doing it: Write applications that use `pma` persistent heaps instead of databases, key-value stores, or files accessed via `read/write`. Persistent memory should simplify applications by eliminating separate formats for ephemeral and persistent data.

Bits

Download `pma` and example code from https://queue.acm.org/downloads/2022/Drill_Bits_07_example_code.tar.gz or http://web.eecs.umich.edu/~tpkelly/pma/, and `pm-gawk` from https://github.com/ucy-coast/pmgawk or https://coast.cs.ucy.ac.cy/projects/pmgawk/.

Drills

1. Rewrite the program in figure 2 to replace persistent memory with `gdbm` or an SQL database. Weigh the pros and cons of each approach.
2. DRAM backed by swap ordinarily underlies virtual memory, but `pma`'s persistent memory sits atop DRAM backed by *storage* beneath the file system. Which has greater capacity on typical systems?
3. Use `pm-gawk`'s persistent heap instead of text files to transport model parameters from the training phase to the filtering phase of Steven Hauser's spam filter.[4]
4. Is eight-byte allocation alignment adequate? See Kuszmaul.[9]

5. Profile an application's use of `malloc`, `free`, etc. using `ltrace`. (Good job for `AWK`:) Write a little simulator to explore how the application's peak memory footprint would vary based on allocator overhead.
6. Research prototypes have integrated both persistence and crash tolerance into JavaScript and Scheme interpreters.[3,13] Would this work equally well for `pm-gawk`?
7. Currently `pma` borrows a handful of time-tested tricks from the heyday of serial allocators.[14] What else could it incorporate without compromising simplicity?
8. Clear (i.e., zero-ize) de-allocated memory on a heavily used persistent heap using `pma_set_avail_mem`. Release storage resources beneath the heap file via "`fallocate --dig-holes`." Measure the savings using the `filefrag` utility.
9. Add persistence to your favorite scripting language interpreter.
10. Design a persistent mutex for use with `pma`.
11. (MENSA members only) Read about the intersection of persistence, parallelism, and crash tolerance in the PMDK book.[11] What's different with `pma`?

Acknowledgments

Programming Pearls author Jon Bentley and `gawk` maintainer Arnold Robbins reviewed an early draft of this column. Robbins furthermore made helpful suggestions about the `pma` interface and about minor changes for the implementation.

References

1. Arpaci-Dusseau, R., Arpaci-Dusseau, A. 2018. *Operating Systems: Three Easy Pieces*. Chapter 29: Lock-based

Concurrent Data Structures. https://pages.cs.wisc.edu/~remzi/OSTEP/.
2. The GNU Allocator. 2022; https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html.
3. Harnie, D., De Koster, J., Van Cutsem, T. SchemeKen: A crash-resilient Scheme interpreter. GitHub; https://github.com/tvcutsem/schemeken.
4. Hauser, S. 2022. Free Unix shell statistical spam filter and whitelist; http://216.92.26.41/article/Statistical_spam_filter.html.
5. Kelly, T. 2019. Persistent memory programming on conventional hardware. *acmqueue* 17(4); https://dl.acm.org/doi/pdf/10.1145/3358955.3358957.
6. Kelly, T. 2020. Hand-over-hand locking for highly concurrent collections. *;login:* 45(3); https://www.usenix.org/system/files/login/articles/login_fall20_14_kelly.pdf.
7. Kelly, T. 2021. Crashproofing the original NoSQL data store. *acmqueue* 18(4); https://dl.acm.org/doi/pdf/10.1145/3487019.3487353.
8. Kelly, T. 2020. Is Persistent Memory Persistent? *Communications of the ACM* 63(9); https://dl.acm.org/doi/pdf/10.1145/3397882.
9. Kuszmaul, B. C. 2015. SuperMalloc: a super-fast multithreaded malloc for 64-bit machines. In *Proceedings of the International Symposium on Memory Management*; http://dx.doi.org/10.1145/2754169.2754178.
10. Lea, D. 2000. A memory allocator; http://gee.cs.oswego.edu/dl/html/malloc.html and http://gee.cs.oswego.edu/pub/misc/malloc.c.
11. Scargall, S. 2020. *Programming Persistent*

Memory. Apress; https://link.springer.com/content/pdf/10.1007/978-1-4842-4932-1.pdf.

12. Tan, Z. F., Li, J., Volos, H., Kelly, T. 2022. Persistent scripting. Non-Volatile Memory Workshop (NVMW); http://nvmw.ucsd.edu/program/.
13. Van Ginderachter, G. v8-ken: A crash-resilient JavaScript interpreter. GitHub; https://github.com/supergillis/v8-ken.
14. Wilson, P. R., Johnstone, M. S., Neely, M., Boles, D. 1995. Dynamic storage allocation: a survey and critical review. In *Proceedings of the International Workshop on Memory Management*; https://dl.acm.org/doi/10.5555/645647.664690. For PDF: https://users.cs.northwestern.edu/~pdinda/ics-s05/doc/dsa.pdf.

Give him a place to stand, and Terence Kelly *<tpkelly@acm.org> will move the Earth. Actually, Zi Fan and Jianan and Haris will do most of the moving; Kelly will write a paper about it.*

CONTENTS