Persistent Scripting

Zi Fan Tan Jianan Li Haris Volos Terence Kelly
Contact: hvolos01@ucy.ac.cy tpkelly@eecs.umich.edu

Persistence High & Low, Inside & Out

Low-level languages have long dominated persistent memory. From early NVM research [1, 14] to PMDK [13], mainstream persistent memory programming has meant C/C++, largely because low-level languages can readily use new CPU instructions needed to fully exploit NVM hardware. Sadly, low-level languages reduce programmer productivity.

High-level scripting languages trade hardware efficiency and control for convenience and productivity by doing what the programmer intends without requiring her to say it. For example, they waive variable declarations and take an easygoing attitude toward data types. Can scripting languages support persistence in the same carefree spirit?

Currently they do not. Python's several persistence options, for example, are fussy and verbose [8, 12, 15]. Worst of all, they explicitly connect *individual* data structures (e.g., associative arrays) to persistence mechanisms. Perl's alternatives are similarly overt and fine-grained.

Process checkpoint/restore suffers the opposite problems: It is too transparent and coarse-grained. It can freeze a running interpreter and re-start it later, like a SIGSTOP/SIGCONT that spans machine reboots. Implementing persistent scripting with CRIU [2], however, taught us that it's wrong for our purposes: Because the interpreter can't perceive resurrection, scripts must include ugly code to find the current run's inputs.

The Right Stuff

To motivate persistent scripting, consider the AWK script below, distilled from a pervasive practical problem.

```
{         # executes once per input line
  if ( ! ($0 in id) )  # assign numeric IDs to
      id[$0] = ++n;    #   unique strings
  freq[$0]++;          # count string frequencies
}
END {     # executes after all input processed
  print n;        # number of unique strings
  for (s in id)   # print table of IDs & frequencies
      print id[s], s, freq[s];
}
```

Text strings, one per input line, appear in variable `$0`. Associative arrays `id` and `freq` assign serial numbers to strings and count their popularity. The script prints a summary table of unique strings with their IDs and frequencies. One of us (Kelly) relied on scripts like this for his dissertation research on Web caching; the strings were URLs from log files.

Log processing, a classic scripting chore, often requires persistence because it is incremental: Every day a new log file arrives and a script like the one above must summarize all logs received to date. For most real-world logs, naïvely processing from scratch the entire archive of N logs on day N is far less efficient than maintaining a persistent summary of all past logs and updating it incrementally as new ones arrive.

In Python or Perl we could use one of the persistence mechanisms discussed earlier, e.g., `perltie` could bind each array in a Perl script to its own `dbm` database. But then the script balloons and spawns a gaggle of `dbm` files, and lone scalars like `n` in the script above are a bother. Alternatively, a script can roll its own persistence, e.g., our AWK script can ingest yesterday's summary before reading today's input log if we prepend

```
BEGIN {  # executes before first input line is read
  getline n < "summary";
  while (0 < (getline < "summary")) {
      id[$2] = $1;  freq[$2] = $3;  }
}
```

But this bloats the code and adds parsing overhead.

The style of persistence closest to the spirit of scripting simply preserves programmer-defined variables across script executions: The next time it runs, a script should awaken prepopulated with all variables that existed when the previous execution terminated. If persistence is implemented as an interpreter option, scripts remain oblivious to persistence. If scripts and their persistent state are stored in separate files, such state may be shared among different scripts.

Persistent Memory `gawk`

We modified the GNU AWK interpreter, `gawk`, to support persistent scripting. Although not today's trendiest scripting language, AWK remains in widespread use [5, 11]. It is supremely convenient for the data-processing tasks for which it was designed and can be remarkably fast [3, 9]. GNU AWK is actively maintained; recent improvements include support for high-precision arithmetic, a profiler, and a debugger. The source code is tidy and maintainer Arnold Robbins explained aspects related to our work.

Persistent memory `gawk` (`pm-gawk`) uses a new persistent memory allocator, `pma`, that allocates from a file-backed memory mapping. The media beneath the filesystem containing the backing file are unconstrained, i.e., `pma` supports persistent memory programming on conventional hardware [6] as well as NVM. The interface is simple and familiar: In addition to the `malloc`/`free` of standard C, `pma` exposes an `init` function that specifies a backing file and `get`/`set` functions that access a *root pointer* from which all live data on `pma`'s persistent heap must be reachable. Under the hood `pma` coalesces freed objects where possible, so any sequence of `pma` calls that releases every allocated object returns `pma` to its initial state; this "reversibility" property greatly aids debugging. Like AWK and `gawk`, `pma` is single-threaded and thus avoids the complexity of parallelism, relying instead on time-tested techniques from the heyday of serial allocators. A companion paper describes `pma` internals and provides source code [7].

Integrating `pma` to create `pm-gawk` was remarkably easy. One reason is `pma`'s conventional "pointerish" `malloc`/`free` interface, which is far more compatible with existing C code

than the "offsettish" allocators of relocatable persistent heaps [6]. We made three changes to `gawk`: First, new command-line option "`--persist=heap.pma`" supplies the backing file containing `pma`'s persistent heap. Second, we replaced calls to conventional `malloc`, `calloc`, `realloc`, and `free` with their `pma` counterparts via `#defines`. Finally, we arranged for the interpreter's dynamically allocated internal data structures representing script-defined variables to be reachable from `pma`'s root pointer. The first two changes were simple. Root-pointer reachability was not much harder, requiring a handful of changes to just one source file, because `gawk`'s internal symbol table provides a single entry point to all script variables for the root. Our changes amount to under 100 lines of `gawk`'s 91,000 LOC. We are working with the `gawk` maintainer to merge our changes into the official distribution; meanwhile a `pm-gawk` fork is available [10].

To use `pm-gawk`, first create for the persistent heap a sparse file whose size is a multiple of the system page size; try "`truncate -s 4096000000 heap.pma`" on the command line. Pass this uninitialized backing file to the `pm-gawk` interpreter via the "`--persist`" flag when executing an AWK script; thereafter the file will contain a persistent heap holding all script variables, which will be available to the script whenever it is run again with `--persist`. For the AWK script of the previous page, the net effect is to render the `BEGIN` block unnecessary: Every run of the script will start with variables `id`, `n`, and `freq` as they were left by the previous run. Worried that an untimely crash could corrupt the persistent heap while it is being modified in-place by a `pm-gawk` script? Make a storage-efficient backup of the heap file before executing the script ("`cp --reflink heap.pma heap.bak; sync`").

The benefits of persistent `gawk` extend beyond persistence. `pm-gawk` is Big Data AWK: Because `pma` allocates memory backed by a file, the size of its heap is limited by available *storage* beneath the filesystem, which often far exceeds available swap. Furthermore, persistent heaps containing script variables may be exchanged freely between different scripts. Heap portability enabled us, for example, to streamline an off-the-shelf spam filter implemented as a pair of AWK scripts [4]. Its model-training script formerly communicated parameters to its filtering script via intermediate text files. With `pm-gawk`, the parameters travel as AWK variables in a persistent heap; writing/parsing intermediate files becomes unnecessary.

Dirty Page *Not* Dirt Cheap

We measure performance of a log-processing workload. For each of 100 simulated days we generate a log file of random strings using a non-stationary distribution to mimic the "hot set drift" observed in real Web access logs. We process 1 billion lines of these logs using the AWK code of the previous page.

Tests N and B use unmodified `gawk`. Test N gobbles all 100 input files at once (the naïve approach). Test B uses the `BEGIN` block to incrementally update its summary of all inputs seen to date. Test P uses `pm-gawk` instead of the `BEGIN` block. For B and P we report run times for the final (hundredth) day only. All script outputs are written to an SSD-backed filesystem; P tests vary the media beneath the persistent heap. Run times reflect in-memory activity only. Time to `sync` data to durability is reported separately because it is off the critical path of data processing.

	time (sec)			speedup vs. N	
test	run	`sync`	total	run	total
N (naïve)	669.43	1.50	670.93	1.00	1.00
B (`BEGIN`)	49.17	1.51	50.68	13.62	13.24
P `/dev/shm/`	53.58	1.51	55.09	12.49	12.18
P Optane block	58.68	23.54	82.22	11.41	8.16
P SSD block	58.77	43.93	102.71	11.39	6.53
P Optane DAX	174.81	3.15	177.96	3.83	3.77

Our results mostly confirm expectations. Incremental processing (B and P/DRAM) is over 10× faster than the naïve approach (N). Incremental processing with `pm-gawk` is roughly as fast as the manual approach (B versus P run times) if the persistent heap resides in DRAM. Optane in DAX mode is slower than DRAM—though the comparison isn't entirely fair because only the former provides durability. Pushing the persistent heap from DRAM to durable media (`sync` times) is faster for Optane configured as a block device than for an SSD, as we would expect.

The *absolute magnitudes* of the `sync` times to block storage, however, are surprisingly large. Why? The `for` loop in the `END` block of our AWK script—conceptually a read-only operation—modifies nearly every memory page of the heap. This does no harm to a conventional heap, but for a persistent heap in a file-backed memory mapping, more dirty pages means more work for `sync`. Persistent memory, whether backed by NVM or block storage, penalizes STORE instructions more severely than conventional ephemeral memory.

Conclusions

Persistent scripting eliminates one of the few remaining undersimplifications of high-level scripting languages by providing the right kind of persistence with zero programmer effort. A `malloc`-compatible persistent heap makes it easy to implement persistent `gawk`, whose in-memory performance is on par with manual persistence. Reducing interpreter STOREs would reduce durability overheads.

References

[1] J. Coburn et al. NV-Heaps. In *ASPLOS*, 2011. [LINK].
[2] https://criu.org/Main_Page.
[3] A. Drake. Command-line Tools can be 235x Faster than your Hadoop Cluster, Jan. 2014. [LINK].
[4] S. Hauser. Unix shell statistical spam filter, Mar. 2022. [LINK].
[5] B. Hoyt. The State of the AWK, May 2020. [LINK].
[6] T. Kelly. Persistent memory programming on conventional hardware. *ACM Queue*, 17(4), July/August 2019. [LINK].
[7] T. Kelly et al. Persistent memory allocation. *ACM Queue*, 20(2), March/April 2022. [LINK].
[8] M. Lutz. *Programming Python*. O'Reilly, 2011. p. 1303.
[9] B. O'Connor. Don't MAWK AWK..., Sept. 2010. [LINK].
[10] Persistent memory `gawk` (`pm-gawk`). https://coast.cs.ucy.ac.cy/projects/pmgawk/ https://github.com/ucy-coast/pmgawk.
[11] Debian popularity contest, Feb. 2022. [LINK].
[12] https://pynvm.readthedocs.io/en/v0.3.1/.
[13] S. Scargall. *Programming Persistent Memory*. Apress, 2020. .
[14] H. Volos et al. Mnemosyne. In *ASPLOS*, 2011. [LINK].
[15] D. Waddington et al. PyMM. In *PLOS*, Oct. 2021. [LINK].