Persistent Memory Programming on Conventional Hardware

The persistent memory style of programming can dramatically simplify application software.

Terence Kelly

The Brave New World of Non-volatile Memory

After years of anticipation in the computer industry, byte-addressable NVM (non-volatile memory) now promises to augment or supplant volatile DRAM (dynamic RAM) in mainstream computers.⁷ Like the ferrite core memory of the 1950s and 1960s, today's NVM retains data even when powered off. Though slower than DRAM, NVM is much faster than conventional durable media—HDDs (hard disk drives) and flash SSDs (solid-state drives).¹⁰ Together with its high density and low cost per byte, NVM's speed and non-volatility will revolutionize memory/storage hierarchies designed around fast low-capacity volatile memory devices and slow high-capacity durable storage devices.¹²

The most exciting opportunity afforded by NVM is radically simplifying how applications access *persistent data*, which must survive process termination and machine reboots. Today's applications manipulate persistent data in block-addressed storage devices via complex layers of intermediate software, such as file systems and databases, whose interfaces and semantics differ fundamentally from those of memory. Application developers must therefore mentally context-switch between divergent programming paradigms (e.g., imperative algorithms using LOAD and STORE instructions to access pointer-rich data structures in memory versus declarative SQL operating upon tables in a relational database). The conceptual "impedance mismatch" between programming styles for ephemeral and persistent data often increases development cost and impairs correctness, maintainability, and performance.¹³

NVM offers the option of storing persistent application data in memory, obviating the need for separate persistence technologies with distinct rules and idiosyncrasies. NVM-only programming brings dramatic simplifications by eliminating complex black-box middleware, eliminating serializers and parsers that translate between memory and storage formats, and allowing the

programmer to think in the single paradigm of in-memory data structures and algorithms. These simplifications may in turn improve cost, correctness, and performance. NVM requires new fault-tolerance mechanisms, but extensive library support and documentation are available.^{8,15}

Of course, most computers today lack NVM and won't be retrofitted with it. Fortunately, the style of programming that NVM invites can be practiced, and many of its benefits obtained, on conventional computers without NVM hardware. This article describes programming techniques and idioms for *persistent memory*, a software abstraction that supports the same simplifications as NVM and yet can be implemented on conventional hardware. These techniques are known among NVM researchers but are neither well documented for developers nor widely taught in computer science curricula.

This article begins by describing a simple and versatile way to practice the persistent memory style of programming on ordinary computers and operating systems, illustrating major points with working C and C++ programs for GNU/Linux (in the following section). The simple approach to persistent memory programming doesn't protect the integrity of persistent application data from corruption by power outages or software crashes, so enhancements that tolerate such failures are examined in the section titled "Tolerating Failures." Finally, the article concludes with prospects for better support and more widespread use of persistent memory programming. All example code is available in machine-readable form at https://queue.acm.org/downloads/2019/Kelly_sample.tar.gz

Ye Olde Persistent Memorie

Persistent memory programming occupies a fourth category of knowledge overlooked by former Secretary of Defense Donald Rumsfeld, famous for his "known knowns," "known unknowns," and "unknown unknowns"; it's something you know, but you don't know that you know it. Persistent memory programming in C/C++ on a Unix-like operating system begins with a straightforward composition of well-known techniques: You simply lay out application data structures (i.e., `structs`) in a file-backed memory mapping, using a pointer cast to interpret the return value of `mmap()` as an entry point into these data structures. Memory mapping allows the application to update persistent data in the backing file via CPU instructions (LOAD and STORE) to the file's in-memory image. For added convenience a *persistent heap* abstraction is typically layered atop raw persistent memory. Despite the familiarity of the basic mechanisms, awareness of this amalgam of techniques among developers is not commensurate with its advantages.

As a warm-up exercise to recall the operation of `mmap()`, consider the C program in figure 1, which trivially interprets a file as a character array and converts every occurrence of the lowercase string "abc" to uppercase "ABC." Performance-conscious readers may compare this in-place approach with the command-line utility alternative,

```
"sed 's/abc/ABC/g' < input > output,"
```

on random text (e.g., from `"openssl rand -base64 10000000 > input"`). Both the in-place transformation of figure 1 and the shell utility must *read* every byte of the input file. The former, however, can be much more frugal with *writes*: Whereas `sed` writes as many bytes as it reads, the `mmap()` approach writes to the backing file only those pages of the in-memory image that have changed—possibly far less data.

```

1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <sys/mman.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6
7  int main(int argc, char *argv[]) {
8      int fd; char *start, *end, *p; size_t size; struct stat sb;
9      if (2 != argc) return 1;
10     if (0 > (fd = open(argv[1], O_RDWR))) return 2;
11     if (0 != fstat(fd, &sb)) return 3;
12     size = (size_t)sb.st_size;
13     start = (char *)mmap(NULL, size, PROT_READ | PROT_WRITE,
14                          MAP_SHARED, fd, 0);
15     if (MAP_FAILED == start) return 4;
16     end = start + size - 2;
17     for (p = start; p < end; p++)
18         if ('a' == *p && 'b' == *(1+p) && 'c' == *(2+p)) {
19             *p = 'A'; *(1+p) = 'B'; *(2+p) = 'C';
20         }
21     return 0; /* without msync(), backing file updated lazily */
22 }
```

FIG 1 In-place string substitution

Like conventional programming, persistent memory programming typically involves a variety of typed, dynamically allocated data structures that embed references such as pointers and iterators. Figures 2 and 3 present a C++ program that illustrates these features. Figure 2 contains a minimalist persistent memory allocator. The key data structure is the header at line 18, which contains allocator bookkeeping information as well as a *root pointer* from

which all in-use persistent application data must be reachable. The header will occupy the first few machine words of the backing file; the remainder of the backing file will contain the persistent heap. The `malloc()` replacement at line 25 does not support recycling freed memory; the corresponding `free()` at line 34 performs sanity checks but is otherwise a no-op. Global operators `new` and `delete` are overloaded to use the simple allocator starting at line 38, ensuring that the STL (Standard Template Library) `<map>` created in figure 3 will reside in persistent memory. The example code cuts corners for brevity and clarity; a production-quality persistent memory allocator would recycle freed memory, and a sophisticated application might create custom container allocators rather than overloading `new` and `delete`.

The code in figure 3 uses the persistent memory allocator of figure 2 to create a persistent STL `<map>` container (line 63) to count the frequency of input words. Because the `<map>` internally contains conventional absolute-address pointers to objects on the persistent heap, the backing file must always be mapped at the same virtual address (variable `"madd"`). Command-line arguments specify the backing file and an optional mapping address. The latter is needed only when creating a new persistent heap; otherwise, the mapping address is obtained from the backing file itself. The `mmap()` flag `MAP_FIXED` is optional (line 73): Without it, the `madd` parameter is a hint that `mmap()` ignores only with good reason (e.g., conflict with an existing mapping). With `MAP_FIXED`, the new mapping might clobber an existing one.

A new persistent heap is initialized starting at line 76: The header is updated to contain the correct mapping address, allocator metadata are initialized, and a new `<map>` is allocated from the persistent heap and made reachable from the root pointer. The application obtains the `<map>`'s entry point from the root pointer at line 85. Thereafter the application may employ the persistent `<map>` as it would any container: traversing it with an iterator (line 90) and updating its contents.

Figure 4 displays a simple C-shell script to compile and run the program of figures 2 and 3. The `truncate` utility creates an empty and initially sparse file to contain the persistent heap; the size of the file is a multiple of the system page size (4,096 bytes). The "pay as you go" storage footprint of sparse files works nicely with persistent heaps: Even if its logical size is enormous, a sparse file consumes storage resources only for nonzero pages. The example program runs twice, each time being fed a few words on `stdin`, then dumping its contents. The first time the program runs, you must supply an address to map the backing file; the second time the address is obtained from the backing file. The output of the second invocation shows that the persistent `<map>` has retained the counts of the input words from the first invocation.

```

1 // Save to file "ex_2_map.c++" and compile as C++98.
2
3 #include <iostream>          // C++ stuff
4 #include <map>
5 #include <cassert>          // C stuff
6 #include <cerrno>
7 #include <cstring>
8 #include <cstdio>
9 #include <cstdlib>
10 #include <fcntl.h>          // POSIX stuff
11 #include <unistd.h>
12 #include <sys/mman.h>
13 #include <sys/stat.h>
14 #include <sys/types.h>
15
16 using namespace std;
17
18 struct Pheap { // header of persistent heap file
19     void *Addr, // address where file should be mapped
20     *End, // points to Addr + size of mapping
21     *Avail, // first byte of memory that can be allocated
22     *Root; // all live data must be reachable from root
23 } *Pheap;
24
25 static void * M(size_t n) { // "bump pointer" allocator
26     char *r = (char *)Pheap->Avail, *p = r + n;
27     if (p > Pheap->End) return NULL; // out of memory
28     while (0 != (p - (char *)0) % sizeof *Pheap)
29         ++p; // align on multiple of header size
30     Pheap->Avail = p;
31     return r;
32 }
33
34 static void F(void * p) { // replaces free(); doesn't recycle
35     assert(p && Pheap && 1 + Pheap <= p && p < Pheap->End);
36 }
37
38 // overload global operators new/delete to use custom allocator
39 bad_alloc e;
40 void * operator new      (size_t s) throw (bad_alloc)
41     { void *r = M(s); if (!r) throw (e); return r; }
42 void * operator new[]    (size_t s) throw (bad_alloc)
43     { void *r = M(s); if (!r) throw (e); return r; }
44 void operator delete     (void * p) throw () { F(p); }
45 void operator delete[]   (void * p) throw () { F(p); }

```

FIG 2 Persistent C++ STL <map>, part 1: Persistent memory allocator

```

46 #define BAIL do { cerr << "BAIL @ line " << __LINE__ << ": " << \
47     strerror(errno) << endl; exit(EXIT_FAILURE); } while (0)
48
49 static size_t filesize(int fd) {
50     struct stat sb; if (0 != fstat(fd, &sb)) BAIL;
51     return (size_t)sb.st_size;
52 }
53 static void * s2add(char *s) {
54     void *r; if (1 != sscanf(s, "%p", &r)) BAIL;
55     return r;
56 }
57 static void * fd2add(int fd) {
58     struct Pheap ph; if (sizeof ph != read(fd, &ph, sizeof ph)) BAIL;
59     return ph.Addr;
60 }
61
62 int main(int argc, char *argv[]) {
63     map<string, int> *STLmap; // we'll make this persistent
64     if (2 != argc && 3 != argc) { cerr << "usage: " << argv[0]
65         << " heapfile [madd]" << endl; return 1; }
66     int fd = open(argv[1], O_RDWR); if (0 > fd) BAIL;
67     size_t fs = filesize(fd);
68     assert(sizeof *Pheap < fs); // heap file too small
69     void *madd = (3 == argc ? s2add(argv[2]) : fd2add(fd));
70     assert(0 == ((char *)madd - (char *)0) % 4096); // page align
71     cerr << "map '" << argv[1] << "' @ " << madd << endl;
72     Pheap = (struct Pheap *)mmap(madd, fs, PROT_READ | PROT_WRITE,
73         MAP_SHARED /* | MAP_FIXED */, fd, 0);
74     if (MAP_FAILED == Pheap) BAIL;
75     if (madd != Pheap) BAIL;
76     if (NULL == Pheap->Addr) { // first time: initialize
77         assert(! Pheap->End && ! Pheap->Avail && ! Pheap->Root);
78         Pheap->Addr = Pheap;
79         Pheap->End = (char *)Pheap + fs;
80         Pheap->Avail = 1 + Pheap;
81         Pheap->Root = new map<string, int>;
82     }
83     else assert(Pheap==Pheap->Addr && Pheap->Avail && Pheap->Root
84         && (char *)Pheap + fs == Pheap->End);
85     STLmap = (map<string, int> *)Pheap->Root; // get root
86     string w;
87     while (cin >> w) {
88         if ("[dump]" == w)
89             for (map<string, int>::iterator it = STLmap->begin();
90                 it != STLmap->end(); ++it) // traverse map
91                 cout << it->first << " : " << it->second << endl;
92         else if (STLmap->find(w) == STLmap->end()) (*STLmap)[w] = 1;
93         else (*STLmap)[w]++;
94         /* if using FAMS, call here ... */
95     }
96     /* ... or here */
97     return 0;
98 }

```

FIG 3 Persistent C++ STL <map>, part 2: Term frequency counter

```

1  #!/bin/csh -f
2  set b = ex_2_map ; rm -f $b
3  g++ -std=c++98 -ansi -pedantic -Wall -Wextra -O3 -o $b $b.c++
4  rm -f $b.ph ; truncate -s 4096000 $b.ph # make new sparse file
5  echo '----- initialize empty persistent heap'
6  echo 'foo bar [dump]' | ./$b $b.ph 0x3fb1d70a0000 # edit if necessary
7  echo '----- run again to verify persistence'
8  echo 'qux foo [dump]' | ./$b $b.ph
9  rm -f $b $b.ph # clean up after ourselves

```

```

1  ----- initialize empty persistent heap
2  map 'ex_2_map.ph' @ 0x3fb1d70a0000
3  bar : 1
4  foo : 1
5  ----- run again to verify persistence
6  map 'ex_2_map.ph' @ 0x3fb1d70a0000
7  bar : 1
8  foo : 2
9  qux : 1

```

FIG 4 Using the persistent <map>: Run script and its output

How do you select an appropriate virtual address for mapping the backing file containing the persistent heap (the magic hex constant in figure 4)? There's not yet a standard portable way to do it; neither POSIX nor Linux currently offer sufficient control of virtual address spaces. The relevant standards and documentation provide no way to guarantee that, for example, shared libraries won't in the future occupy parts of the address space where you placed a persistent heap in the past. In practice, however, a simple heuristic suffices to identify a "quiet neighborhood" in an address space where you can reliably map a persistent heap: Simply find the largest unused address range obtainable via `mmap()` and place the persistent heap near the middle of that range. The simple C program of figure 5 is one way to find such an empty address range; inspecting `/proc/self/maps` is another. Line 16 of figure 5 avoids a subtle bug.² The address that is passed to `mmap()` must be aligned on a page boundary (figure 3, line 70), and if you also pass the `MAP_FIXED` flag, then you must first verify that the specified address range is empty (not shown in figure 3).


```

1  #define _GNU_SOURCE /* for non-standard mmap() flags */
2  #include <stdio.h>
3  #include <sys/mman.h>
4
5  #define MMAP(L) mmap(NULL, (L), PROT_NONE, MAP_PRIVATE \
6                      | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0)
7  #define MUNMAP(A, N) do { if (0 != munmap((A), (N))) \
8                          return 1; } while (0)
9
10 int main(int argc, char *argv[]) {
11     void *A, *Amax = NULL; size_t L = 0, U, Max = 0;
12     for (U = 1; ; U *= 2) /* double upper bound until failure */
13         if (MAP_FAILED == (A = MMAP(U))) break;
14         else MUNMAP(A, U);
15     while (1 + L < U) { /* binary search between bounds */
16         size_t M = L + (U - L) / 2; /* avoid overflow */
17         if (MAP_FAILED == (A = MMAP(M))) { U = M; }
18         else { Amax = A; Max = M; MUNMAP(A, M); L = M; }
19     }
20     printf("%lu bytes at %p\n", Max, Amax);
21     return 0;
22 }

```

FIG 5 Finding a large gap in the virtual address space

Consistently mapping a persistent heap at a constant address allows you to use conventional C-style pointers in familiar and convenient ways. As the program of figures 2 and 3 demonstrates, it also allows the reuse of existing pointer-based libraries by simply sliding a persistent heap beneath them. Constant-address heaps thus enable persistent memory programming to leverage vast amounts of existing software designed for ordinary ephemeral memory. Inflexible mapping, however, has downsides beyond those already mentioned. Sharing data between independently managed applications becomes difficult: If by chance two people have both created persistent heap files that must be mapped at overlapping virtual address ranges, they can't use both files simultaneously in a single process.

To summarize, if a persistent heap is to be private to a process in the same sense that a conventional heap is private, then mapping at a constant address may be a small price to pay for the advantages of absolute pointers. If persistent data are to be shared across independent applications, however (i.e., if the data resemble a database rather than a heap), then relocatable data structures are appropriate.

Relocatable persistent data structures employ relative *offsets* instead of absolute-address pointers. Offsets are computed with respect to the base address at which the backing file is mapped, which is now permitted to vary

from one run of a program to the next. Replacing pointers with offsets eliminates the need to map persistent data consistently at a constant address and makes it easy to share persistent data among different applications and users. The downside of offsets is that they're incompatible with existing pointer-based libraries (e.g., C++ STL), and they lack the familiarity, convenience, and type safety of pointers.

Figures 6 and 7 present a C program that builds a binary search tree atop a relocatable persistent heap. This program, like the previous C++ example, counts the frequency of input words. The generic persistent heap infrastructure of figure 6 is broadly similar to that of figure 2, but a mapping address is no longer stored in the header (line 18), and function `o2p()` is added to convert offsets to pointers (line 43). Figure 7 contains a homebrew binary search tree that allocates relocatable offset-based data structures from the persistent heap of figure 6. Manually coding offset-based data structures is usually more annoying than this simple example suggests; conversion between offsets and pointers disrupts the natural flow of programming. Furthermore, the necessary pointer casts compromise type checking. Carefully crafted C++ libraries could hide much of the fuss of offsets, as in the leading NVM-specific library collection.⁸ While syntactic sugar can make offset-based programming seem more natural, the loss of compatibility with existing pointer-based software remains a steep price to pay for relocatability.

Regardless of whether offsets or pointers are used, persistent memory programming imposes a handful of new rules and guidelines. Avoid references from persistent memory to ordinary ephemeral memory (e.g., a `struct` allocated on a persistent heap containing a pointer to the stack or to the conventional heap). Such references are a fertile source of bugs because persistent memory typically outlives volatile memory; global, external, and static variables can cause problems for similar reasons. C++ must be used with care lest objects contain invisible function pointers (`vtbl`).

Multithreading brings additional rules. Mutexes embedded in persistent data aren't reset across program restarts; better to embed *indices* into a mutex array in *ephemeral* memory, because the latter can be reset easily at process startup. For synchronization primitives requiring more elaborate initialization, more sophisticated reset mechanisms have been designed.⁹ During an `msync()` call no thread may modify the in-memory image; otherwise, a data race would occur, which is illegal in C, C++, and POSIX, and which would leave the backing file in an indeterminate state.

```

1  /* Save to file "ex_3_reloc_tree.c" and compile as C99. */
2
3  #include <assert.h>
4  #include <errno.h>
5  #include <fcntl.h>
6  #include <inttypes.h>
7  #include <stdio.h>
8  #include <string.h>
9  #include <unistd.h>
10 #include <sys/mman.h>
11 #include <sys/stat.h>
12 #include <sys/types.h>
13
14 #define UNIT sizeof(uint64_t) /* unit of pmem allocation, alignment */
15
16 typedef uint64_t pmem_offset_t; /* dist from mmap addr in UNITS */
17
18 typedef struct { /* header of persistent heap file */
19     pmem_offset_t avail, end, /* memory allocator bookkeeping */
20     root; /* entry point to application data */
21 } pmem_header_t;
22
23 static pmem_header_t *pmem_header;
24
25 static void pmem_init(const size_t len) {
26     assert(0 == sizeof *pmem_header % UNIT); /* simplifying assumption */
27     assert(0 == len % UNIT);
28     assert(0 == pmem_header->avail && /* verify p-heap is new */
29            0 == pmem_header->end && 0 == pmem_header->root);
30     pmem_header->avail = sizeof *pmem_header / UNIT;
31     pmem_header->end = len / UNIT;
32 }
33
34 static pmem_offset_t pmem_alloc(const size_t n) {
35     pmem_offset_t r = pmem_header->avail,
36     N = n / UNIT + (0 == n % UNIT ? 0 : 1);
37     if (pmem_header->avail + N > pmem_header->end)
38         return 0; /* offset analog of NULL pointer; out of memory */
39     pmem_header->avail += N;
40     return r;
41 }
42
43 static void * o2p(pmem_offset_t off) { /* convert offset to ptr */
44     return 0 == off ? NULL : (char *)pmem_header + off * UNIT;
45 }

```

FIG 6 Relocatable persistent search tree, part 1: Persistent heap

```

46 typedef struct { uint64_t N;          /* term frequency counter */
47                  pmem_offset_t L, R; /* "pointers" to children */
48                  char word[];        /* illegal C89, okay C99 */
49 } treenode_t;
50
51 static void tree_dump(pmem_offset_t node) { /* traverse in-order */
52     if (0 != node) {
53         treenode_t *p = (treenode_t *)o2p(node);
54         tree_dump(p->L);
55         printf("%s : %" PRIu64 "\n", p->word, p->N);
56         tree_dump(p->R);
57     }
58 }
59
60 static pmem_offset_t tree_insert(pmem_offset_t node, char *s) {
61     treenode_t *p = (treenode_t *)o2p(node);
62     if (0 == node) { /* allocate a new node */
63         pmem_offset_t nu = pmem_alloc(sizeof *p + 1 + strlen(s));
64         treenode_t *np = (treenode_t *)o2p(nu);
65         assert(0 != nu); /* allocation failure */
66         np->N = 1;
67         np->L = np->R = 0;
68         strcpy(np->word, s);
69         return nu;
70     }
71     if (0 == strcmp(s, p->word)) ++(p->N); /* found word */
72     else if (0 > strcmp(s, p->word)) p->L = tree_insert(p->L, s);
73     else p->R = tree_insert(p->R, s);
74     return node;
75 }
76
77 int main(int argc, char *argv[]) {
78     int fd; struct stat s; char buf[10240]; size_t fsize;
79     if (2 != argc) {
80         fprintf(stderr, "usage: %s heapfile\n", argv[0]); return 1; }
81     if (-1 == (fd=open(argv[1], O_RDWR))) { perror("open"); return 2; }
82     if (0 != fstat(fd, &s)) { perror("fstat"); return 3; }
83     fsize = (size_t)s.st_size;
84     pmem_header = (pmem_header_t *)
85     mmap(NULL, fsize, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
86     if (MAP_FAILED == pmem_header) { perror("mmap"); return 4; }
87
88     if (0 == pmem_header->avail) pmem_init(fsize);
89     while (1 == scanf(" %s", buf)) {
90         if (0 == strcmp("[dump]", buf))
91             tree_dump(pmem_header->root);
92         else
93             pmem_header->root = tree_insert(pmem_header->root, buf);
94         /* if using FAMS, call here ... */
95     }
96     /* ... or here */
97     return 0;
98 }

```

FIG 7 Relocatable persistent search tree, part 2

Persistent memory programming is prone to new risks that developers must mitigate. Because application software manipulates persistent data directly, the blast radii of the usual C/C++ bugs enlarge. Persistent memory also exacerbates the so-called "schema evolution" problem: If application requirements change, dictating that a new field must be added to an existing `struct`, then extensive changes to the persistent heap will be required because data structures are packed tightly in the heap. Developers may avert or postpone this problem by padding data structures with unused space in anticipation of future expansion, but in the worst case they must write reformatting code. Reformatting code must also be written if a persistent heap is migrated to new hardware with, for example, a different pointer size or different endian-ness.

The biggest problem with `mmap()`-based persistent memory is that failures such as power outages, operating-system kernel panics, and application process crashes threaten to corrupt persistent data. Standard POSIX `mmap()` and its implementations on Unix-like operating systems permit the operating system to write modified pages of the in-memory image to the backing file at any time and in any order. A crash during updates to persistent data may therefore leave the backing file in an *inconsistent* state that violates application-level invariants and correctness criteria, making recovery impossible.

Tolerating Failures

An update mechanism that operates atomically even in the presence of failures can prevent untimely crashes from corrupting persistent data. Applications may then evolve persistent data atomically from one consistent state to the next; following a crash, recovery is guaranteed to begin with consistent data. *Failure-atomic* `msync()` (FAMS) guarantees that the backing file underlying a memory mapping always reflects the most recent successful `msync()`, regardless of crashes. FAMS is well suited to `mmap()`-based persistent memory programming because it merely restricts the behavior of a standard system call and is therefore easy to reason about. FAMS is also well suited to conventional hardware: Whereas NVM-specific atomicity mechanisms track and log memory modifications at machine-word or cache-line granularity,^{3,4,8,19} FAMS operates upon memory pages, typically 4 KiB, similar to the update granularity of block-addressed storage devices.

Failure-atomic `msync()` tolerates fail-stop failures such as power outages, operating-system kernel panics, and application process crashes. It's not magic; hardware or software bugs that directly corrupt the backing file are not tolerated. Bugs that corrupt the in-memory image cannot corrupt the backing

file unless FAMS is called. If a program detects corruption in memory before calling FAMS and then refrains from calling it, the integrity of the backing file is unaffected. Therefore, it is wise to check application-level invariants and correctness criteria as thoroughly as possible before calling FAMS.

The rules for using FAMS in persistent memory programming are easy to master: If and only if persistent application data occupying the in-memory image of a file-backed memory mapping are consistent, then the programmer may call failure-atomic `msync()`. For example, permissible locations for FAMS calls are indicated with comments in figure 3 (lines 94 and 96) and figure 7 (lines 94 and 96). If a FAMS implementation is used to memory map a file, the backing file is updated only when FAMS is called, so a FAMS call is needed at one of the indicated locations in the example programs. Historically, failure to pair calls properly—`malloc()/free()`, `lock()/unlock()`, `open()/close()`—has been a fertile source of bugs. Fortunately, FAMS does not involve pairs of "begin transaction" / "end transaction" calls; every interval between consecutive FAMS calls defines an atomic bundle of updates. Finally, in multithreaded programs, as with conventional `msync()`, no thread may modify the in-memory image during a FAMS call.

The classic example of transferring money between bank accounts shows when FAMS may (not) be called and illustrates the prudent practice of checking invariants:

```
1. sum = acct[A] + acct[B];
2. acct[A] -= 10;
3. failure_atomic_msync();           /* bug! */
4. acct[B] += 10;
5. assert(acct[A] + acct[B] == sum);
6. failure_atomic_msync();           /* correct */
```

Calling FAMS between the decrement of account A and the increment of account B is a bug that allows a crash (e.g., between lines 3 and 4) to destroy money, because the "conservation of money" invariant does not hold. In contrast, it is permissible to call FAMS after incrementing account B, which restores the invariant.

Though it's overkill for the simple example above, checking invariants before calling FAMS is strongly recommended in general. It's safe to use ordinary assertions because if FAMS is used correctly, the overall program can tolerate a crash—or an assertion failure—at any point in its execution. For veteran developers, one of the hardest aspects of the FAMS learning curve is "crashing with confidence." Programmers who embrace the safe crashes enabled by FAMS can often write simpler software by avoiding corner-case recovery and exception code (see Yoo et al., Section 3.3,²¹ for guidelines and examples).

Failure-atomic `msync()` can be implemented in at least three ways: in an operating system, a file system, or a user-space library. FAMS is completely flexible with respect to the underlying durable storage system because none of these implementations imposes constraints or requirements on the storage layer; storage is a placeholder. Options include individual HDDs or SSDs, networked storage arrays, and georeplicated cloud storage—even NVM devices configured to expose a block storage interface. This flexibility allows users and operators of FAMS-based applications to tailor storage choices to requirements of cost, performance, capacity, availability, and other considerations without any changes to the application. For library implementations that store data in files, the same flexibility applies to the choice of file system.

Since 2011 my colleagues and I have implemented FAMS several times and have deployed it in two commercial products. The first implementation was embedded in a fault-tolerant distributed computing library called Ken.^{11,21} Ken's user-space FAMS later became the foundation of crash tolerance in Hewlett-Packard's Indigo line of high-volume printing presses.¹ Our next FAMS implementation was a prototype in the Linux kernel.¹⁴ HP's Storage Division implemented a generalization of FAMS in AdvFS (Advanced File System), a Linux file system used in storage appliance products.¹⁷ A group in academia independently implemented FAMS in a file system designed for hybrid NVM/DRAM memories.²⁰

Because storage devices and crash-tolerance mechanisms have often fallen short of designer aspirations and user expectations,^{22,23} my colleagues and I tested our FAMS implementations extensively. We instrumented critical parts of the source code with "crashpoints," which allow test frameworks to methodically crash the software between every pair of semicolons. We also injected crashes externally at random times, both via `kill -ABRT` as well as by injecting kernel panics using an interface exposed by the `/proc/` pseudo-file system on Linux. Most importantly, all of the implementations were subjected to sudden whole-system power interruptions, which stress the entire hardware/software stack more strenuously than any other kind of test. The powerfail tests yielded by far the most useful and interesting information. For example, one round of tests involving five SSDs revealed that two lost data and one became an expensive paperweight as a result of power cutoffs.¹⁴ Regardless of the crash-tolerance mechanism used, the right maxim for developers is *train as you would fight*: Testing should subject the entire hardware/software stack to the full range of failures that it must tolerate in production.

Extensive experience has shown that it can be remarkably easy to retrofit FAMS-based crash tolerance onto complex software that was not designed for FAMS. Changing *one* line of code in the Kyoto Tycoon key-value server sufficed to upgrade its non-crash-tolerant mode of operation to a crash-safe mode based on Linux-kernel FAMS; furthermore, FAMS-based crash tolerance proved to be

markedly faster than Tycoon's native transaction mechanism.¹⁴ Performing a similar upgrade to SQLite's non-crash-tolerant mode of operation using the FAMS implementation in AdvFS required changing *two* lines of code and improved performance by an even greater margin.¹⁸ Sliding a user-space FAMS beneath the complex, performance-optimized software controlling HP Indigo printing presses required modest effort from a single developer and reduced crash recovery times from days to minutes.¹ A simple persistent heap layered atop FAMS allowed other developers to retrofit crash safety onto a Scheme interpreter⁵ and a JavaScript interpreter.¹⁶

The performance of FAMS depends heavily on the implementation (operating system, file system, or library) and underlying durable storage media; detailed performance evaluations are presented in the references cited above. Fortunately, it's easy to reason about the fundamentals of FAMS performance even without knowing which kind of storage system will provide durability: The performance cost of a FAMS call is approximately a constant overhead plus an additional cost proportional to the number of memory pages modified since the previous call. Any implementation must perform a fixed amount of work on every FAMS call (the constant overhead) and per modified memory page (the linear term). Performance-conscious programmers avoid gratuitous page modifications and call FAMS sparingly.

The sweet spot of FAMS performance is similarly easy to characterize: The overall size of persistent data should be large; if it's small, the incremental operation of FAMS is overkill and the simple expedient of reading a file, changing it in memory, and writing it to a new file suffices. The fraction of memory pages changed between consecutive FAMS calls should be small; otherwise, the incrementality of FAMS is unhelpful and you might as well simply write the entire in-memory image to storage. The number of pages modified between FAMS calls should be large; otherwise, the fixed cost of a FAMS call might dominate the linear term. Finally, the number of modifications to each modified memory page should be large, because it would be wasteful to pay the overhead of logging just to flip a single bit on a page. Like many other aspects of computing and programming, FAMS-based persistent memory rewards access locality.

At least two new implementations of FAMS are in progress at the time of this writing. The first is an implementation in the XFS file system.⁶ Like the earlier HP AdvFS implementation of FAMS,¹⁷ the XFS implementation promises to be more efficient than a user-space library because it operates directly upon file-system data structures and thereby avoids the double write of logging. The XFS implementation is essentially complete, and thorough testing is planned.

The second ongoing FAMS implementation is a library called `famus` (failure-atomic `msync()` in user space) that I am writing. Whereas my earlier user-space FAMS¹¹ detected modified memory pages by catching SIGSEGVs (segmentation violation signals) and used REDO logging for atomicity, `famus` uses Linux "soft dirty bits" to detect memory modifications and UNDO logging for atomicity. A side benefit of UNDO logging in a user-space FAMS is data versioning: Every FAMS call defines a version of the application's persistent data, and previous versions can be recovered by applying UNDO logs to the backing file. If versioning is not needed, each UNDO log may be deleted upon successful completion of the FAMS call that created it. Regardless of whether logs are retained or deleted, the backing file always contains the most recent FAMS-committed state of persistent data.

Optimizations reduce the size of `famus` logs. In general, `famus` performs two durable page writes for each memory page modified between consecutive FAMS calls: The prior version of the page is written from the backing file to the UNDO log, and the modified version of the page is written from memory to the backing file. `Famus` optimizes away the page write to the UNDO log in two situations: First, if the page contains only zero bytes, the log simply records the page ID and flags it as an empty page, omitting 4 KiB of zeros; second, if the "modified" page is byte-for-byte identical to its state at the previous FAMS call, the page may be safely ignored.

The first optimization helps as an initially sparse backing file is gradually filled with nonzero pages; without the first optimization, the UNDO log would contain many zero pages. The second optimization applies to a surprisingly widespread but seldom noted phenomenon: STORE instructions that overwrite memory with identical contents. For example, realistic inputs cause nearly every call to `tree_insert()` in figure 7 to overwrite a word of memory with the same value that was already there. When programs such as this are run with `famus`, the second optimization substantially reduces the size of UNDO logs.

At the time of this writing I believe `famus` to be complete and correct, though it has not yet been tested as thoroughly as previous FAMS implementations. A preliminary release is available at <http://web.eecs.umich.edu/~tpkelly/famus/>

Summary and Prospects

Together with a handful of readily mastered techniques and idioms, `mmap()` supports a hardware-agnostic software abstraction of persistent memory. When practiced tastefully and with sound judgment, the persistent memory style of programming can dramatically simplify application software. If crash tolerance

is required, the failure-atomic `msync()` abstraction admits a wide range of implementations, all of which offer unrestricted flexibility with respect to the underlying durable storage media. Experience has shown that FAMS can make it remarkably easy to retrofit crash tolerance onto complex legacy software. At least four FAMS implementations have been developed, two of which have been incorporated into commercial products. Two new implementations are nearing completion.

Prior work has extensively but not exhaustively explored FAMS implementations. A new user-space design might track memory modifications with the existing Linux `userfaultfd()` interface; a Linux implementation of the Windows `GetWriteWatch()` interface would be a better starting point. Another possible foundation for FAMS is per-file snapshotting via `ioctl(FICLONE)` in file systems such as Btrfs and XFS. Stronger semantics for `mlock()` on file-backed memory mappings would greatly facilitate a user-space FAMS on Unix-like operating systems. Finally, a sophisticated persistent memory allocator designed for the FAMS performance cost model—pay \$1 per page modified between `msync()` calls—remains to be written.

Given the convenience and compatibility of conventional pointers, better support is needed for memory mappings at application-specified addresses. Enabling the confident and portable use of such mappings requires only a standardized consensus that application mappings have top priority on a reserved region of the address space; linkers, conventional memory allocators, and other components in need of unused address space merely agree to use this region only as a last resort, allowing applications to occupy it first. When relocatable data structures are required, the Microsoft C/C++ language extension of "based pointers" is helpful; similar features in Unix compilers would be a welcome improvement. Improved compiler support for controlling virtual address layout would also help. For example, programmers should be able to coerce page alignment for variables and arrays, making it easy to map files under them.

The persistent memory style of programming would be as useful for type- and memory-safe languages as for C/C++. Future efforts might draw inspiration from research prototypes of crash-tolerant interpreters for Scheme⁵ and JavaScript,¹⁶ which enable applications to employ ordinary variables rather than external databases for persistent data. JavaScript is a particularly promising candidate because of its event-driven nature; the handling of each event is simply made into a mini-transaction (e.g., by concluding it with a failure-atomic `msync()` call).

Driven by the advent of byte-addressable non-volatile memory, the persistent memory style of programming will gain traction among developers, taking its rightful place alongside existing paradigms for managing persistent application

state. Until NVM becomes available on all computers, developers can use the techniques presented in this article to enjoy the benefits of persistent memory programming on conventional hardware.

References

1. Blattner, A., Dagan, R., Kelly, T. 2013. Generic crash-resilient storage for Indigo and beyond. Technical Report HPL-2013-75, Hewlett-Packard Laboratories; <http://www.hpl.hp.com/techreports/2013/HPL-2013-75.pdf>.
2. Bloch, J. 2006. Extra, extra—read all about it: nearly all binary searches and mergesorts are broken. Google AI Blog (June); <https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html>.
3. Chakrabarti, D., Boehm, H., Bhandari, K. 2014. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the ACM International Conference on Object-oriented Systems Language and Applications (OOPSLA)*, 433-452; <https://dl.acm.org/citation.cfm?id=2660224>; <http://www.hpl.hp.com/techreports/2013/HPL-2013-78R1.pdf>.
4. Coburn, J., Caulfield, A., Akel, A., Grupp, L., Gupta, R., Jhala, R., Swanson, S. 2011. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 118; <https://dl.acm.org/citation.cfm?id=1950380>.
5. Harnie, D., De Koster, J., Van Cutsem, T. SchemeKen: a crash-resilient Scheme interpreter; <https://github.com/tvcutsem/schemeken>.
6. Hellwig, C. 2019. Failure-atomic file updates for Linux. Forthcoming in Linux Piter (October); <https://linuxpiter.com/en/materials/2307>; patches: <https://www.spinics.net/lists/linux-xfs/msg04536.html> and http://git.infradead.org/users/hch/vfs.git/shortlog/refs/heads/O_ATOMIC.
7. Intel. Intel Optane Technology; <http://www.intel.com/optane/>.
8. Intel. Persistent Memory Development Kit; <http://pmem.io/pmdk/>.
9. Intel. 2017. Synchronization primitives for NVM. <http://pmem.io/2016/05/31/cpp-08.html>; <http://pmem.io/2015/06/18/threads.html>.

10. Izraelevitz, J., Yang, J., Zhang, L., Memaripour, A., Soh, Y. J., Dulloor, S. R., Zhao, J., Kim, J., Liu, X., Wang, Z., Xu, Y., Swanson, S. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. (April); <https://arxiv.org/abs/1903.05714>.
11. Kelly, T. Ken: a platform for fault-tolerant distributed computing; <http://ai.eecs.umich.edu/~tpkelly/Ken/>.
12. Nanavati, M., Schwarzkopf, M., Wires, J., Warfield, A. 2015. Non-volatile storage. *ACM Queue* 13(9); <https://queue.acm.org/detail.cfm?id=2874238>.
13. Neward, T. 2006. Object/Relational Mapping: The Vietnam of computer science. Ted Neward's Blog; <http://blogs.tedneward.com/post/the-vietnam-of-computer-science/>.
14. Park, S., Kelly, T., Shen, K. 2013. Failure-atomic msync(): a simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 225-238; <https://dl.acm.org/citation.cfm?id=2465374>; http://web.eecs.umich.edu/~tpkelly/papers/Failure_atomic_msync_EuroSys_2013.pdf.
15. Rudoff, A. 2017. Persistent memory programming. *login*; 42(2), 34-40; https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf.
16. Van Genderachter, G. V8Ken: a crash-resilient JavaScript interpreter; <https://github.com/supergillis/v8-ken> and http://web.eecs.umich.edu/~tpkelly/Ken/V8Ken_value_proposition.html
17. Verma, R., Mendez, A. A., Park, S., Mannarswamy, S., Kelly, T., Morrey, B. 2015. Failure-atomic updates of application data in a Linux file system. In *Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST)*; <https://www.usenix.org/system/files/conference/fast15/fast15-paper-verma.pdf>.
18. Verma, R., Mendez, A. A., Park, S., Mannarswamy, S., Kelly, T., Morrey, B. 2015. SQLearn: database acceleration via atomic file update. Technical Report HPL-2015-103, Hewlett-Packard Laboratories; <http://www.labs.hpe.com/techreports/2015/HPL-2015-103.pdf>.
19. Volos, H., Tack, A. J., Swift, M. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 91-104; <https://dl.acm.org/citation.cfm?id=1950379>.

20. Xu, J., Swanson, S. 2016. NOVA: a log-structured file system for hybrid volatile/nonvolatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*; <https://www.usenix.org/system/files/conference/fast16/fast16-papers-xu.pdf>.
21. Yoo, S., Killian, C., Kelly, T., Cho, H. K., Plite, S. 2012. Composable reliability for asynchronous systems. Usenix Annual Technical Conference (ATC); <https://www.usenix.org/system/files/conference/atc12/atc12-final206-7-20-12.pdf>.
22. Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E. S., Zhao, B. W., Singh, S. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th Usenix Symposium on Operating Systems Design and Implementation (OSDI)*; https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-zheng_mai.pdf. (Note that an errata sheet is provided separately.)
23. Zheng, M., Tucek, J., Qin, F., Lillibridge, M. 2013. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th Usenix Conference on File and Storage Technologies (FAST)*; <https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf>.

Related articles

Research for Practice

[Distributed Consensus and Implications of NVM on Database Management Systems](#)

Expert-curated guides to the best of CS research
<https://queue.acm.org/detail.cfm?id=2967618>

The Morning Paper: [SageDB and NetAccel](#)

Learned models within the database system; network-accelerated query processing
Adrian Colyer
<https://queue.acm.org/detail.cfm?id=3317289>

Terence Kelly <tpkelly@acm.org> is a Distinguished Member and a Lifetime Member of the ACM. He earned a Ph.D. in computer science at the University of Michigan, Ann Arbor, in 2002. Kelly spent 14 years at Hewlett-Packard Laboratories; during his last five years at HPL he developed software support for non-volatile memory. Kelly now teaches and evangelizes persistent memory programming. His patents and publications are listed at <http://ai.eecs.umich.edu/~tpkelly/papers/>.