Persistent Memory Programming on Conventional Hardware

Terence Kelly
tpkelley@eecs.umich.edu
http://ai.eecs.umich.edu/~tpkelley/

PIRL, UC San Diego
22 July 2019

© 2019 Terence Kelly, all rights reserved.
NVM Returns
Advantage: Simplicity

- Fewer moving parts: no quirky external persistent store
- Single data format: no translators, serializers, parsers
- Single paradigm: no mental context switching, “impedance mismatch”
- Often improves cost, correctness, performance
Persistent Memory

- *Software* abstraction
- Corresponding programming style
- Similar to NVM-style programming, some of same advantages
- Implementable on conventional hardware, without NVM
- This talk: C/C++ on POSIX/Linux
Persistent Memory
Persistent Memory: Elements

- Application data in file-backed memory mapping
- Sparse backing files: pay-as-you-go
- Pointer casts interpret file as application data types
- Persistent heap allocates from file-backed memory mapping
- Sliding persistent heap beneath legacy software
mmap()
Warm-Up: In-Place “abc” → “ABC”

```
char *start, *end, *p;

/* trivially interpret file as byte array */
start = (char *)mmap(NULL, filesize, PROT_READ | PROT_WRITE,
    MAP_SHARED, filedescriptor, 0);

end = start + filesize - 2;
for (p = start; p < end; p++) { /* access persistent data... */
    if ('a' == *p && 'b' == *(p+1) && 'c' == *(p+2)) {
        *p = 'A';
        *(p+1) = 'B'; /* ... and STOREs */
        *(p+2) = 'C';
    }
} /* cf. sed */
```
struct foo { int bar; double qux; } *foop;

/* interpret file contents as application-defined type */
foop = (struct foo *)mmap(NULL, filesize,
    PROT_READ | PROT_WRITE,
    MAP_SHARED, filedescriptor, 0);

/* access/update persistent data ... */
if (foop->bar) /* ... via LOADs ... */
    foop->qux *= 2.0; /* ... and STOREs, with type checking */
typedef struct { void *mapaddr, *root, *avail, *end; } *pheap_t; /* persistent heap bookkeeping */
static pheap_t e_pheap;

static void *M(size_t s) { /* simple bump-pointer allocator */
    void *r = e_pheap->avail;
    size_t u = sizeof *e_pheap, nu = s / u + (0 == s % u ? 0 : 1);
    e_pheap->avail = (pheap_t)(e_pheap->avail) + nu;
    assert(e_pheap->avail <= e_pheap->end); /* out of memory? */
    return r;
}

static void F(void *p) { assert(NULL != p); return; } /* just leak it */
#define BA bad_alloc
#define OP operator
#define NU (size_t s) throw (BA) { void *r = M(s); if (!r) throw (e); return r; }
#define DL (void * p) throw ( ) { F(p); }

BA e;

void * OP new NU void OP delete DL /* overload global operators */
void * OP new[] NU void OP delete[] DL
void *ma; /* address to map backing file */
...
ssize_t n = read(backing_file_fd, &ma, sizeof ma); /* is address already in backing file? */
if (NULL == ma) {
    /* find good address range to map backing file; famus provides code */
}

e_pheap = (pheap_t)mmap(...);

map<string, int> *m; /* entry point to persistent C++ STL <map> */

if (NULL == e_pheap->mapaddr) {
    /* initialize newborn persistent heap */
    e_pheap->mapaddr = e_pheap; /* map at same addr next time */
    e_pheap->avail = 1+e_pheap;
    e_pheap->end = (char *)e_pheap + s;
    m = new map<string, int>;
    e_pheap->root = m; /* root ptr set to entry persistent <map> */
}
else { /* resume existing persistent heap */
    m = (map<string,int>*)e_pheap->root; /* find <map> via root */
}

/* preliminaries over; use the <map> as usual; below is input word frequency counter */
while (cin >> w) {
    if ("dump" == w)
        for (map<string, int>::const_iterator it = m->begin(); it != m->end(); ++it)
            cout << "dump: " << it->first << " " << it->second << endl;
    else if (m->find(w) == m->end()) (*m)[w] = 1;
    else (*m)[w]++;
}
Is the `<map>` Persistent?

```bash
% truncate -s 409600 backf  # create empty sparse file

% (echo foo; echo bar; echo '[dump]') | ./ex_02 backf
dump: bar 1
dump: foo 1

% (echo qux; echo foo; echo '[dump]') | ./ex_02 backf
dump: bar 1
dump: foo 2
dump: qux 1
```
Fixed vs. Relocatable

Map persistent data at same address every time

+ allows conventional pointers: familiar, convenient
+ retrofit persistence onto legacy libraries, programs

− not standard/portable (but works in practice)
− inhibits sharing between independent programs

■ use for private heaps, existing code

Make persistent data structures relocatable

+ facilitates sharing persistent data among independent apps
+ more standard/portable

− requires offsets: less convenient/natural than pointers
− incompatible with pointer-based existing code

■ use for shared databases
Avoid bugs due to lifetime mismatches
- pointers from persistent to conventional memory
- static, external, global variables
- C++ vtbl

Data races remain illegal
- don’t modify in-memory image during msync()

Re-initialize mutexes etc. following re-start
- PMDK has a sophisticated way
Failures

- Power outages, OS kernel panics, application process crashes
- May corrupt backing file
- Conventional `mmap()`/`msync()` not atomic w.r.t. failure
- POSIX & default Linux filesystems don’t support efficient standard failure-atomic update for *application data* in files
- Filesystems protect only *metadata*
  - cf. police prevent crime against cops
- Constructing atomicity from ordering primitives (fsync(), msync()) left as an exercise
What!? No Failure Atomicity!?
FAMS: Failure-Atomic `msync()`

- Backing file always reflects most recent `msync()`
  - even if failures occur
- Restricts behavior of conventional `mmap()/msync()`
  - easy to reason about
- Efficient
  - like ordinary `msync()`, writes only dirty pages
- Tolerates fail-stop failures
  - e.g., power outages, OS panic, application crash
  - direct corruption of backing file not tolerated
  - not magic; doesn’t fix C/C++ safety problems
FAMS Rules, Guidelines, Tips

- All rules for conventional `mmap()` / `msync()`
- Call FAMS only when data consistent, invariants hold
- Check invariants carefully before calling FAMS
- Crash with confidence
FAMS Implementations

Past:
- “Ken,” user-space library (USENIX ATC 2012)
- Linux kernel (EuroSys 2013)
- HP Advanced File System (FAST 2015)
- NOVA filesystem (FAST 2016)

Present/Future:
- XFS, Christoph Hellwig
- famus: failure-atomic msync() in user space
Experience

- Tycoon networked key-value store: 1 LOC
- SQLite database: 2 LOC
- Crash-resilient JavaScript interpreter ("V8Ken")
- HP Indigo printing presses

*FAMS makes it easy to retrofit crash resilience*
Flexible implementation: OS, filesystem, library

Storage is a placeholder (for most implementations)
  - local HDD/SSD, RAID array, cloud storage
  - durability, availability, endurance, performance
  - application doesn’t change

Big-data friendly (some implementations)
  - overall persistent data: max file size
  - working set: max virtual memory
  - “transaction size” (Δ between msync()s): free storage

No pairing “begin transaction” / “end transaction”
  - cf. malloc/free, lock/unlock, open/close
FAMS Attractions II

- Easy to retrofit crash resilience onto legacy code
- Easy to reason about: restricts familiar semantics
- Easy to check global invariants: before calling FAMS
- Data versioning for free (in some implementations)
- No special hardware required
famus: Failure-Atomic `msync()` in User Space

- Clean-slate re-write from scratch
- Modification tracking: Linux soft dirty bits
- Atomicity: UNDO logging
- Supports data versioning
  - Every failure-atomic `msync()` call defines a version
  - Retain UNDO logs, apply to backing file to revert
- Preliminary release
  - [http://web.eecs.umich.edu/~tpkelly/famus/](http://web.eecs.umich.edu/~tpkelly/famus/)
  - **WARNING:** No powerfail tests yet
  - performance not optimal
- Questions, support, feedback: tpkelly@eecs.umich.edu
Summary

- Persistent memory is a *software* abstraction
- Style of programming & benefits similar to NVM programming
- Implementable on conventional hardware
- Failure-atomic `msync()` addresses failures
- `famus` implements FAMS on Linux
  - [http://web.eecs.umich.edu/~tpkelly/famus/](http://web.eecs.umich.edu/~tpkelly/famus/)