Peddle the Pedal to the Metal

Howard Chu
CTO, Symas Corp. hyc@symas.com
Chief Architect, OpenLDAP hyc@openldap.org

2019-08-31

HTTP://SYMAS.COM

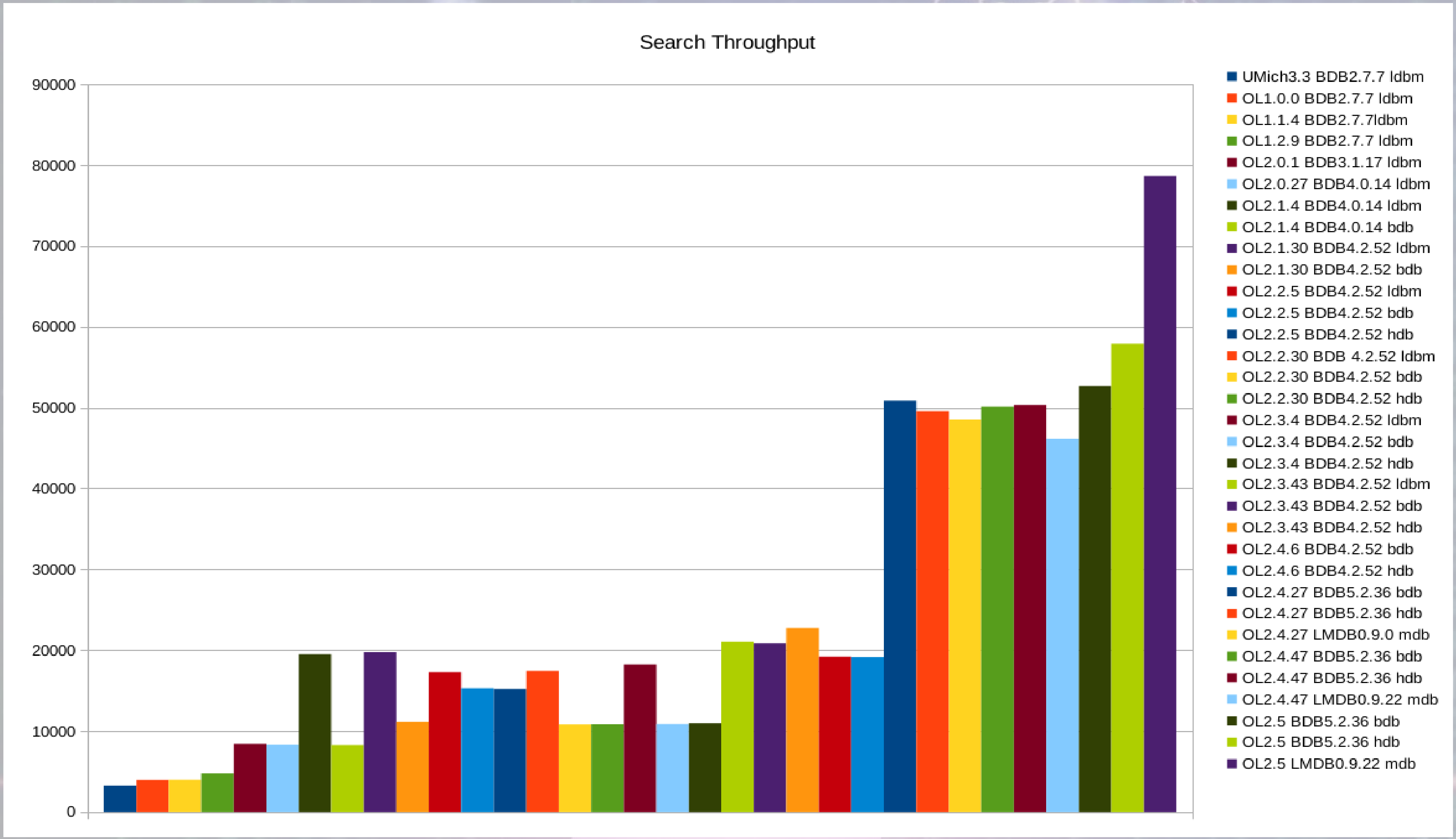
Overview

- Context, philosophy, impact
- Profiling tools
- Obvious problems and effective solutions
- More problems, more tools
- When incremental improvement isn't enough

Tips, Tricks, Tools & Techniques

- Real world experience accelerating an existing codebase over 100x
  - From 60ms per op to 0.6ms per op
  - All in portable C, no asm or other non-portable tricks

Search Performance



Mechanical Sympathy

- “By understanding a machine-oriented language, the programmer will tend to use a much more efficient method; it is much closer to reality.”
  - Donald Knuth *The Art of Computer Programming* 1967

Optimization

- “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”
  - Donald Knuth “Computer Programming as an Art” 1974

Optimization

- The decisions differ greatly between refactoring an existing codebase, and starting a new project from scratch
  - But even with new code, there's established knowledge that can't be ignored.
    - e.g. it's not premature to choose to avoid BubbleSort
    - Planning ahead will save a lot of actual coding

Optimization

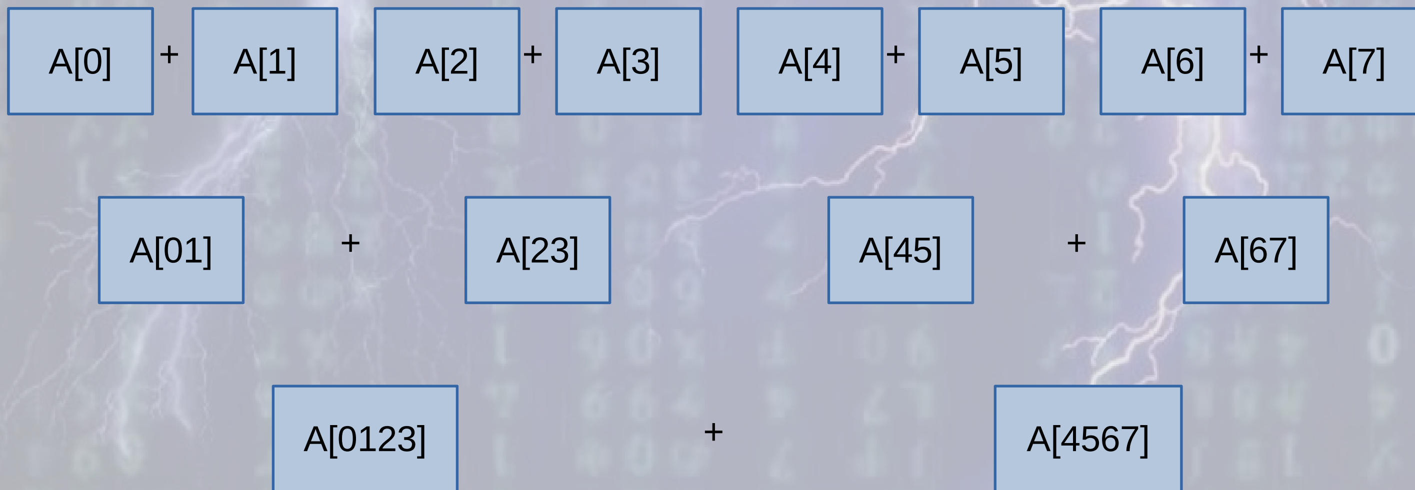
- Eventually you reach a limit, where a time/space tradeoff is required
  - But most existing code is nowhere near that limit
- Some cases are clear, no tradeoffs to make
  - E.g. there’s no clever way to chop up or reorganize an array of numbers before summing them up
    - Eventually you must visit and add each number in the array
    - Simplicity is best

Summing

A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7]

```
int i, sum;
for (i=1, sum=A[0]; i<8; sum+=A[i], i++);
```

Summing




```
int i, j, sum=0;
for (i=0; i<5; i+= 4) {
  for (j=0; j<3; j+=2) a[i+j] += a[i+j+1];
  a[i] += a[i+2];
  sum += a[i];
}
```

Optimization

- Correctness first
  - It's easier to make correct code fast, than vice versa
- Try to get it right the first time around
  - If you don't have time to do it right, when will you ever have time to come back and fix it?
- Computers are supposed to be fast
  - Even if you get the right answer, if you get it too late, your code is broken

Tools

- Profile! Always measure first
  - Many possible approaches, each has different strengths
    - Linux perf (formerly called oprofile)
      - Easiest to use, time-based samples
      - Generated call graphs can miss important details
    - FunctionCheck
      - Compiler-based instrumentation, requires explicit compile
      - Accurate call graphs, noticeable performance impact
    - Valgrind callgrind
      - Greatest detail, instruction-level profiles
      - Slowest to execute, hundreds of times slower than normal

Profiling

- Using `perf` in a first pass is fairly painless and will show you the worst offenders
  - We found in UMich LDAP 3.3, 55% of execution time was spent in malloc/free. Another 40% in strlen, strcat, strcpy
  - You’ll never know how (bad) things are until you look

Profiling

- As noted, `perf` can miss details and usually doesn't give very useful call graphs
  - Knowing the call tree is vital to fixing the hot spots
  - This is where other tools like FunctionCheck and valgrind/callgrind are useful

Insights

- “Don’t Repeat Yourself” as a concept applies universally
  - Don’t recompute the same thing multiple times in rapid succession
    - Don’t throw away useful information if you’ll need it again soon. If the information is used frequently and expensive to compute, remember it
    - Corollary: don’t cache static data that’s easy to re-fetch

String Mangling

- The code was doing a lot of redundant string parsing/reassembling
  - 25% of time in strlen() on data received over the wire
    - Totally unnecessary since all LDAP data is BER-encoded, with explicit lengths
    - Use struct bervals everywhere, which carries a string pointer and an explicit length value
    - Eliminated strlen() from runtime profiles

String Mangling

- Reassembling string components with strcat()
  - Wasteful, Schlemiel the Painter problem
    - https://en.wikipedia.org/wiki/Joel_Spolsky#Schlemiel_the_Painter%27s_algorithm
    - strcat() always starts from beginning of string, gets slower the more it's used
  - Fixed by using our own strcopy() function, which returns pointer to end of string.
    - Modern equivalent is stpcpy().

String Mangling

- Safety note – safe strcpy/strcat:

```
char *stecpy(char *dst, const char *src, const char *end)
{
	while (*src && dst < end)
		*dst++ = *src++;
	if (dst < end)
		*dst = '\0';
	return dst;
}

main() {
	char buf[64];
	char *ptr, *end = buf+sizeof(buf);

	ptr = stecpy(buf, "hello", end);
	ptr = stecpy(ptr, " world", end);
}
```

String Mangling

- stecpy()
  - Immune to buffer overflows
  - Convenient to use, no repetitive recalculation of remaining buffer space required
  - Returns pointer to end of copy, allows fast concatenation of strings
  - You should adopt this everywhere

String Mangling

- Conclusion
  - If you’re doing a lot of string handling, you probably need to use something like struct bervals in your code

```
struct berval {
    size_t len;
    char *val;
}
```

  - You should avoid using the standard C string library

Malloc Mischief

- Most people's first impulse on seeing "we're spending a lot of time in malloc" is to switch to an "optimized" library like jemalloc or tcmalloc
  - Don't do it. Not as a first resort. You'll only net a 10-20% improvement at most.
  - Examine the profile callgraph; see how it's actually being used

Malloc Mischief

- Most of the malloc use was in functions looking like

```
datum *foo(param1, param2, etc…) {
    datum *result = malloc(sizeof(datum));
    result->bar = blah blah…
    return result;
}
```

HTTP://SYMAS.COM

Malloc Mischief

- Easily eliminated by having the caller provide the datum structure, usually on its own stack

```
void foo(datum *ret, param1, param2, etc…)
{
    ret->bar = blah blah...
}
```

HTTP://SYMAS.COM

Malloc Mischief

- Avoid C++ style constructor patterns
  - Callers should always pass data containers in
  - Callees should just fill in necessary fields
- This eliminated about half of our malloc use
  - That brings us to the end of the easy wins
  - Our execution time accelerated from 60ms/op to 15ms/op

Malloc Mischief

- More bad usage patterns:
  - Building an item incrementally, using realloc
    - Another Schlemiel the Painter problem
  - Instead, count the sizes of all elements first, and allocate the necessary space once

Malloc Mischief

- Parsing incoming requests
  - Messages include length in prefix
  - Read entire message into a single buffer before parsing
  - Parse individual fields into data structures
- Code was allocating containers for fields as well as memory for copies of fields
- Changed to set values to point into original read buffer
- Avoid unneeded mallocs and memcpys

Malloc Mischief

- If your processing has self-contained units of work, use a per-unit arena with your own custom allocator instead of the heap
  - Advantages:
    - No need to call free() at all
    - Can avoid any global heap mutex contention
  - Basically the Mark/Release memory management model of Pascal

Malloc Mischief

- Consider preallocating a number of commonly used structures during startup, to avoid cost of malloc at runtime
  - But be careful to avoid creating a mutex bottleneck around usage of the preallocated items
- Using these techniques, we moved malloc from #1 in profile to … not even the top 100.

Malloc Mischief

- If you make some mistakes along the way you might encounter memory leaks
- FunctionCheck and valgrind can trace these but they’re both quite slow
- Use github.com/hyc/mleak – fastest memory leak tracer

Uncharted Territory

- After eliminating the worst profile hotspots, you may be left with a profile that's fairly flat, with no hotspots
  - If your system performance is good enough now, great, you're done
  - If not, you're going to need to do some deep thinking about how to move forward
  - A lot of overheads won't show up in any profile

Threading Cost

- Threads, aka Lightweight Processes
  - The promise was that they would be cheap, spawn as many as you like, whenever
  - (But then again, the promise of Unix was that processes would be cheap, etc…)
  - In reality: startup and teardown costs add up
    - Don’t repeat yourself: don’t incur the cost of startup and teardown repeatedly

Threading Cost

- Use a threadpool
  - Cost of thread API overhead is generally not visible in profiles
  - Measured throughput improvement of switching to threadpool was around 15%

Function Cost

- A common pattern involves a Debug function:

```
Debug(level, message) {
    if (!( level & debug_level ))
        return;
    …
}
```

Function Cost

- For functions like this that are called frequently but seldom do any work, the call overhead is significant
- Replace with a DEBUG() macro
  - Move the debug_level test into the macro, avoid function call if the message would be skipped

Function Cost

- We also had functions with huge signatures, passing many parameters around
- This is both a correctness and efficiency issue
- “If you have a procedure with 10 parameters, you probably missed some.”
  - Alan Perlis *Epigrams on Programming* 1982

Function Cost

- Nested calls of functions with long parameter lists use a lot of time pushing params onto the stack
- Instead, put all params into a single structure and pass pointer to this struct as function parameter
- Resulted in 7-8% performance gain
  - https://www.openldap.org/lists/openldap-devel/200304/msg00004.html

Data Access Cost

- Shared data structures in a multithreaded program
  - Cost of mutexes to protect accesses
  - Hidden cost of misaligned data within shared structures: “False sharing”
    - Only occurs in multiprocessor machines

Data Access Cost

- Within a single structure, order elements from largest to smallest, to minimize padding overhead
- Within shared tables of structures, align structures with size of CPU cache line
  - Use mmap() or posix_memalign() if necessary
- Use instruction-level tracing and cache hit counters with perf to see results

Data Access Cost

- Use mutrace to measure lock contention overhead
- Where hotspots appear, try to distribute the load across multiple locks instead of just one
  - E.g. in slapd threadpool, work queue used a single mutex
  - Splitting into 4 queues with 4 mutexes decreased contention and wait time by a factor of 6.

Stepwise Refinement

- Writing optimal code is an iterative process
  - When you eliminate one bottleneck, others may appear that were previously overshadowed
  - It may seem like an unending task
  - Measure often and keep good notes so you can see progress being made

Burn It All Down

- Sometimes you'll get stuck, maybe you went down a dead end
- No amount of incremental improvements will get the desired result
- If you can identify the remaining problems in your way, it may be worthwhile to start over with those problems in mind

Burn It All Down

- In OpenLDAP, we've used BerkeleyDB since 2000
  - Have spent countless hours building a cache above it because its own performance was too slow
  - Numerous bugs along the way related to lock management/deadlocks
- Realization: if your DB engine is so slow you need to build your own cache above it, you've got the wrong DB engine

Burn It All Down

- We started designing LMDB in 2009 specifically to avoid the caching and locking issues in BerkeleyDB
- Changing large components like this requires a good modular internal API to be feasible
  - Rewriting the entire world from scratch is usually a horrible idea, reuse as much as you can that's worth saving
  - Make sure you actually solve the problems you intend, make sure those are the actual important problems

Burn It All Down

- LMDB uses copy-on-write MVCC, exposes data via read-only mmap
  - Eliminates locks for read operations, readers don’t block writers, writers don’t block readers
  - Eliminates mallocs and memcpy when returning data from the DB
    - There are no blocking calls at all in the read path, reads scale perfectly linearly across all available CPUs
  - DB integrity is 100% crash proof, incorruptible
    - Restart after shutdown or crash is instantaneous

Review

- Correctness first
  - But getting the right answer too late is still wrong
- Fixing inefficiencies is an iterative process
- Multiple tools available, each with different strengths and weaknesses
- Sometimes you may have to throw a lot out and start over

Conclusion

- Ultimately the idea is to do only what is necessary and sufficient
  - Do what you need to do, and nothing more
  - Do what you need, once
  - DRY talks about not repeating yourself in source code; here we mean don't repeat yourself in execution

Questions?