Data-structure lock-in

D. J. Bernstein
University of Illinois at Chicago

The browser is slow

I ran `chromium-browser`
`http://bench.cr.yp.to/results-hash.html`.

Unsurprising: slow load.
This page is 8509794 bytes +
32136149 bytes for 151 pictures.

Surprising: slow search.
`Ctrl-F boris` took *seconds*
to find `boris` on the page.

More searches; same slowness.

du is slow

```
du -s x
```

is a standard UNIX command
showing total space used by
files `x/*`, `x/*/*`, `x/*/*/*`, etc.
(Doesn't follow symlinks.)

du is slow

`du -s x`
is a standard UNIX command
showing total space used by
files `x/*`, `x/*/*`, `x/*/*/*`, etc.
(Doesn't follow symlinks.)

I ran `du -s ~`
on the SSD on my laptop.

`du` is slow

`du -s x`
is a standard UNIX command
showing total space used by
files `x/*`, `x/*/*`, `x/*/*/*`, etc.
(Doesn't follow symlinks.)

I ran `du -s ~`
on the SSD on my laptop.

This was painfully slow:
2 minutes, 42 seconds.

Repeated: *2 minutes, 0 seconds*.

`make` is slow

Typical `make` input:

```
prog: prog.c
	gcc -o prog prog.c
```

If `prog.c` changes,

`make` runs `gcc -o prog prog.c`.

make is slow

Typical make input:

```
prog: prog.c
	gcc -o prog prog.c
```

If prog.c changes,
make runs gcc -o prog prog.c.

After compiling
NVIDIA_GPU_Computing_SDK
I tweaked a few files
and ran make again.

make is slow

Typical make input:

```
prog: prog.c
        gcc -o prog prog.c
```

If prog.c changes,
make runs gcc -o prog prog.c.

After compiling
NVIDIA_GPU_Computing_SDK
I tweaked a few files
and ran make again.

Time for make:
compiler time *plus 15 seconds*.

Why does this happen?

Thousands of papers and books
say how to organize data
in memory; on disk; on networks.

Why does this happen?

Thousands of papers and books
say how to organize data
in memory; on disk; on networks.

Common student exercises
in data-structure design:

1. Keep track of summaries.
2. Keep log of changes.
3. Keep a search index.

Why does this happen?

Thousands of papers and books say how to organize data in memory; on disk; on networks.

Common student exercises in data-structure design:

1. Keep track of summaries.
2. Keep log of changes.
3. Keep a search index.

But real-world programs often fail to apply these exercises. Why?

Case study: LZSS

One way to print

`yabbadabbadabbadoo`:

- print `yabbad`;
- go back 5, copy 4;
- go back 5, copy 5;
- print `doo`.

Case study: LZSS

One way to print
`yabbadabbadabbadoo`:

- print `yabbad`;
- go back 5, copy 4;
- go back 5, copy 5;
- print `doo`.

`yabbad5455doo`
is more concise than
`yabbadabbadabbadoo`.

This is an example of
LZSS decompression.

Typical LZSS compressor:
find longest match
of $\leq$16 bytes within
previous $\leq$4096 bytes;
print position, length.

Typical LZSS compressor:
find longest match
of $\leq$16 bytes within
previous $\leq$4096 bytes;
print position, length.

Programmer starts with
simplest implementation.

Typical LZSS compressor:
find longest match
of $\leq$16 bytes within
previous $\leq$4096 bytes;
print position, length.

Programmer starts with
simplest implementation.

Perhaps language is C.
Programmer uses an array:

```
char buffer[4096+16];
int bufferlen;
int alreadyencoded;
```

Programmer implements operations on this array:

- initialize;
- read more data;
- find longest match;
- move past the match.

Some code; not very complicated.

Programmer implements
operations on this array:

- initialize;
- read more data;
- find longest match;
- move past the match.

Some code;
not very complicated.

Programmer measures speed.
Oops, painfully slow.

Problem #1:
Moving past the match copies the entire buffer, if `alreadyencoded>=4096`.

Problem #1:
Moving past the match
copies the entire buffer,
if `alreadyencoded>=4096`.

Standard solution:
Circular buffer.

Problem #1:
Moving past the match
copies the entire buffer,
if `alreadyencoded>=4096`.

Standard solution:
Circular buffer.

Problem #2, even bigger:
Finding longest match
performs a variable scan
from each buffer position.

Problem #1:
Moving past the match
copies the entire buffer,
if `alreadyencoded>=4096`.

Standard solution:
Circular buffer.

Problem #2, even bigger:
Finding longest match
performs a variable scan
from each buffer position.

Standard solution:
Maintain an index.

These data-structure changes
require *reimplementing*
the data-structure operations.

These operations are
most of the compression code!

These data-structure changes
require *reimplementing*
the data-structure operations.

These operations are
most of the compression code!

Not a huge cost:
this is a simple program.

But what happens when
this cost is scaled
to much larger systems?

Clearly something is going wrong:
Chromium isn't making an index.

Reusable data structures

Easily find implementations
of various data structures.

Some associative-array examples:
`hsearch` in C and
`unordered_map` in C++,
hash tables in memory;
`dbm`/`ndbm`/`sdbm`/`gdbm`,
hash tables on disk;
`db`, memory + disk;
`dir_index` in ext3/ext4;
arrays in `awk`;
`dict` in `python`.

Languages often provide concise syntax for associative arrays, encouraging widespread use.

python: `x['hello'] = 5`

/bin/sh: `echo 5 > x/hello`

But what happens when the programmer needs more than an associative array?

Example: List of events.
Priority-queue operations:
find and remove first event;
add new event.

`heapq` in `python`
supports these operations
but does not support `[...]`.
Incompatible with `dict`:
conversion is easy but slow.

What if programmer receives
a `dict` from a library
and wants its first element?

Can find implementations
of more advanced structures
such as AVL trees,
supporting priority-queue ops
and associative-array ops.

```
d = avltree()
addmystuffto(d)
print d.first()
```

The `addmystuffto` library
can do `d[...]=...`
without knowing whether
`d` is a `dict`, an `avltree`, etc.
"Duck typing."

But Python doesn't
encourage this library design.

`mystuff` library probably
creates its own `dict`:
`d = mystuff()`

Programmer who wants
`avltree` instead of `dict`
then has to modify library
or pay for conversion.

Modifying one library is cheap
but modifying many is not.

Reusable filesystems

UNIX filesystem is a tree.

Each internal node ("directory")
is an associative array
mapping strings to subnodes.

Each leaf node ("file")
is a simple array of bytes.

ext3, UFS, etc.
all provide this API.
Typical applications
work on top of this API.

Good:

Tree structure allows
efficient priority queue
(if directories are small);
finding all `a/b/*`; etc.
Much more powerful than,
e.g., `dict` in `python`.

Good:
Tree structure allows
efficient priority queue
(if directories are small);
finding all `a/b/*`; etc.
Much more powerful than,
e.g., `dict` in `python`.

Bad:
Ad-hoc distinctions between
the tree structure,
the associative arrays,
and the simple arrays.
Too many ways to do one thing.

Good:
Changing the filesystem (switching from ext3 to UFS, adding features to ext3, etc.) doesn’t break normal programs.

Good:
Changing the filesystem
(switching from ext3 to UFS,
adding features to ext3, etc.)
doesn't break normal programs.

Bad:
Extra filesystem operations
are a hassle for programs
to access.

Good:
Changing the filesystem
(switching from ext3 to UFS,
adding features to ext3, etc.)
doesn't break normal programs.

Bad:
Extra filesystem operations
are a hassle for programs
to access.

Even worse:
Changing the filesystem
is a huge deployment hassle.

Speeding up `du -s`
is conceptually straightforward:
modify filesystem to track
`du -s` result for each directory.

Speeding up `du -s`
is conceptually straightforward:
modify filesystem to track
`du -s` result for each directory.

But how does an application
access this result?
New `ioctl`?
Reserve a special filename?

Compare to Python:
new data structure implements
a `totalusage()` function,
immediately usable by caller.
Separate from user namespace.

Even worse: How do we deploy this modified filesystem?

Filesystems are integrated into operating-system kernels. Much harder to modify than per-application code.

Some attempts to do better: loopback NFS, Plan 9, FUSE. But API is still a mess.

Conclusion

Inadequate modularization has locked us into many bad data-structure decisions.

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."
—David L. Parnas, "On the criteria to be used in decomposing systems into modules," 1972