Moving Bottlenecks: CPU Cycle Optimization Using Liboil

David Schleef
<ds@schleef.org>

gstreamer

About Me

Projects:

GStreamer, swfdec, Comedi, ESound

Channels:

#gstreamer, #xorg, #cairo, #gnome-hackers

Hire me

Liboil – what is it?

- “LIBrary of Optimized Inner Loops”
- Collection of simple functions operating on arrays
- Framework for testing and profiling functions

License: 2-clause BSD

inner loops

```
for (y = 0; y < h; y++){
  for(x = 0; x < w; x++){
    pixels[y * rowstride + x] = val;
  }
}
```

```
for (y = 0; y < h; y++){
  oil_splat_u32_ns (pixels + y * rowstride,
      &val, w);
}
```

Function Class

```
oil_splat_u32_ns (uint32_t *d_1xn, uint32_t *s1_1, int n)
  all implementations:
    splat_u32_ns_mmx
      flags: mmx, mmxext
      profile: 547 ticks (std.dev. 1.16667)
      sum abs difference: 0 (n=1000)
      currently chosen
    splat_u32_ns_unroll4
      profile: 1423 ticks (std.dev. 2.73354)
      sum abs difference: 0 (n=1000)
    splat_u32_ns_unroll2
      profile: 2052 ticks (std.dev. 2.62996)
      sum abs difference: 0 (n=1000)
    splat_u32_ns_ref
      flags: REF
      profile: 2042 ticks (std.dev. 1.16667)
```

Types of function classes

- simple math (+, -, *, /,...)
- type conversion (int->float)
- colorspace conversion (YUV->RGB)
- codec (dct, idct, mdct, transpose, zigzag)
- image manipulation (scaling, compositing)
- misc (UTF-8 checking, md5sum)
- ...anything that doesn't involve decisions in code

How liboil works

- oil_init()
  - _oil_cpu_int() -- check CPU capabilities
  - oil_optimize_all()
    - oil_optimize_class()
      - create random input array
      - create reference output array
      - run and profile each implementation
      - compare result to reference
      - choose fastest

(~25 msec startup)

ABI policy

- liboil-0.3 has a stable ABI
- ABI changes every 6-12 months
- add/remove symbols only
- liboil-0.4.x will provide liboil-0.4.so *and* liboil-0.3.so
- new 0.3.so will link against 0.4.so

Purpose: benefits of long-term ABI stability, but still allowing changes

Bottlenecks

- Data
  - HD to main memory (15 MB/s)
  - main memory to CPU (600 MB/s - 2 GB/s)
  - L2 to CPU (4 GB/s)
  - L1 to CPU (8 GB/s)
- Instructions
  - prefetching/cracking
  - register contention
  - data dependencies
  - decision points
  - execution units

Why Optimize?

- profiling shows code is CPU bound
- inherintly CPU bound algorithm
  - small data set
  - high computation cost
- (future-ish) memory speed bound
  - memset/memcpy

The Guilty

- [redacted]
  - byte swapping code that is 1/10 the speed of glibc's memcpy
- [redacted]
  - hand coded memcpy(), plus algorithm that uses 3 copies
- [redacted]
  - “some day it will be fast”

History

- 6 previous attempts

Attempt #1: if (have_mmx)

- very common strategy (kernel, X, libmpeg2)
- problem: not all MMX created equal
- problem: SSE, MMX-ext, SSE2, 3D-Now!

(non-attempt #1.5: CPU-specific .so libs)

Attempt #2: algorithm level

- compile algorithms multiple times, inlining different inner loops
- choose path by CPU capabilities
- problem: *way* too complex
- random CFLAGS and inline assembly code don't play well

Attempt #3: simdpack

- hand-picked palettes of implementations
- hand-written test and profile code
- problem: lots of buggy tests
- problem: lots of buggy implementations
- problem: NFS root file systems

Goals

- Simple (easy to understand)
- Simple (low overhead)
- Integrated testing and profiling
- Any number of implementations per class

Attempt #4: liboil-0.3

- problem: global pointers to functions
- problem: can't add classes

Alternate Approaches

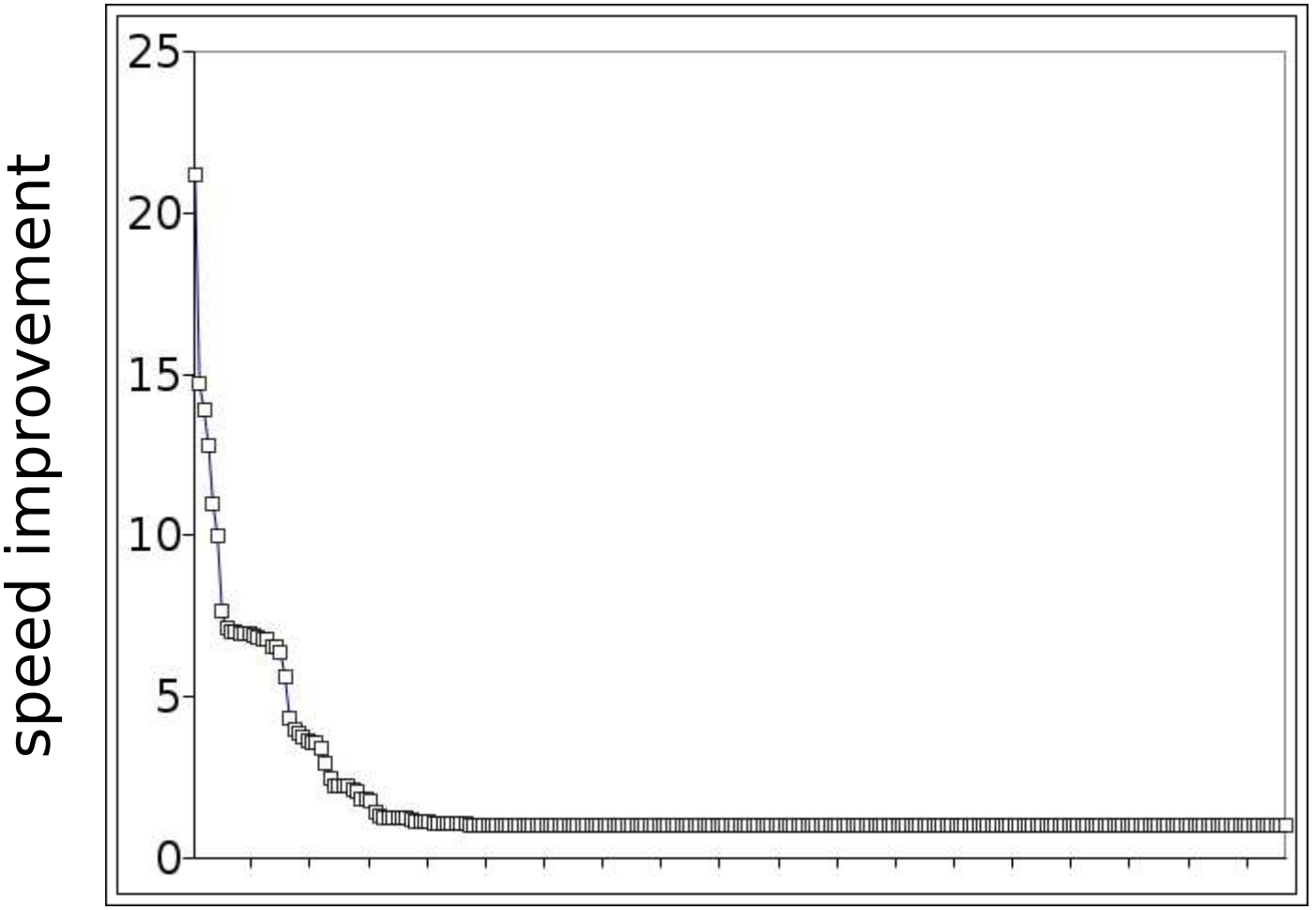
- Separate binaries
  - glibc, fftw
- Custom compilation
  - gentoo, Mplayer
- if(have_mmx)
  - X, Mplayer, kernel, ... (lots)
- Trust the compiler

GCC

- gcc-4.0 vector code generation
  - cool, but poor performance
  - SSE/SSE2 only
- liboil uses any new GCC performance improvements
- GCC can't compete with hand-coded assembly

Performance

(gromit)



gstreamer

random features

- embeddable
- “-O99 -funroll-all-loops -fgo-faster” option

Future Directions

- world domination
- strategic algorithm choice
- handling cache footprint
- überopt
- caching of profile data

Why use liboil?

- liboil has the infrastructure you need anyway
- share difficult-to-maintain common code
- liboil will only get faster
- ABI stable
- no dependencies

Porting existing code

- separate algorithm from inner loops
- adjust inner loops to conform to liboil classes
- add classes to liboil as necessary
- convert existing MMX code to liboil function implementations
- check regression

Porting existing code

- or, cry because algorithm is hopelessly intertwined with loops
- “uniformly slow code”

Getting Involved

- copy existing code into liboil
- write code for !i386, !powerpc
- make GCC better
- compile with icc and copy the assembly
- write better C implementations

Philosophy

- don't spend time on <10% improvement
- decision points are bad
- simple function classes, but don't outrun L1 cache

Moving Bottlenecks: CPU Cycle Optimization Using Liboil

David Schleef
<ds@schleef.org>

gstreamer