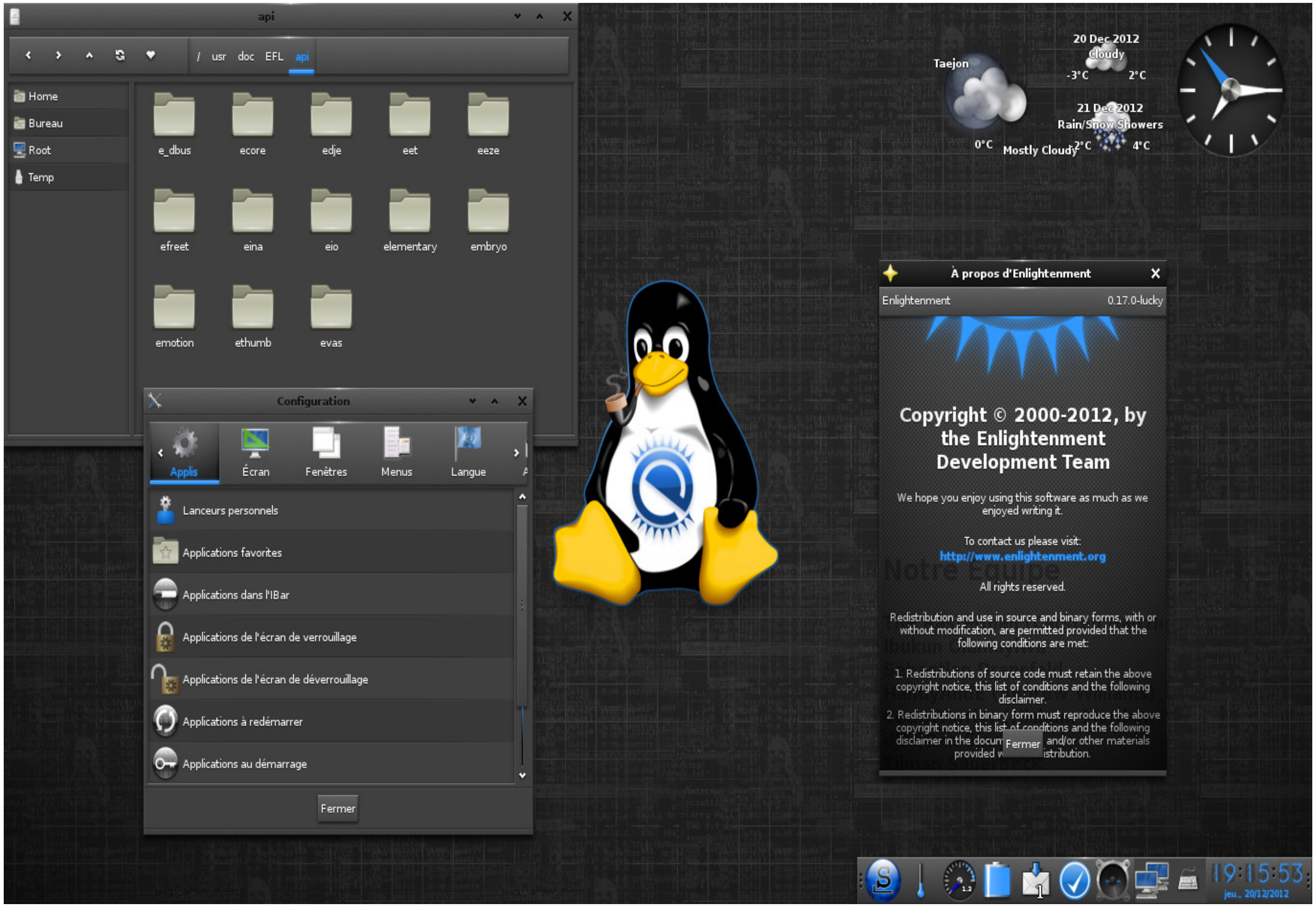
Enlightenment Foundation Libraries

A Case Study of Optimizing for Wearable Devices

Cedric BAIL
Samsung Open Source Group
cedric@s-opensource.com

EFL: A Toolkit Created for Enlightenment 17



Copyright © 2000-2012, by the Enlightenment Development Team

We hope you enjoy using this software as much as we enjoyed writing it.

To contact us please visit:
http://www.enlightenment.org

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the docum and/or other materials provided w istribution.

Enlightenment 17

- Enlightenment project started in 1997
- Window Manager
- First Window Manager of GNOME
- Full rewrite started in 2001
- Primary belief is there will never be *“a year of the Linux desktop”*
- Designed with the embedded world in mind…
- … and needed a toolkit!
- None matched our need back then, this remains true today!

Enlightenment

- 17 was just the version number, we are now at 21!
- Use EFL scenegraph and main loop
- Not different from any other EFL application
- Composite and Window Manager
- Wayland client support
- Multiple backends (X, FB, DRM, …)
- Highly customizable (Profiles, modules and themes)
- Performs better on a Raspberry Pi 3 than Weston

Enlightenment Foundation Libraries (*EFL*)

- The community spent a decade writing a modern graphic toolkit
- Licensed under a mix of LGPL 2.1 and BSD license (Yes, nobody owns it, nobody can change the license)
- Focuses on embedded devices
- Used in Samsung's Tizen products
- First released in January 2011
- Stable, long term API/ABI
- In the process of releasing version 1.19
- ~3 month release cycle

State of EFL

- Originally designed to create a Windows Manager (WM), now used for any type of application
- Has its own scenegraph and rendering library
- Optimized to reduce CPU, GPU, memory, and battery usage
- Supports international language requirements (LTR/RTL, UTF8)
- Supports all variations of screens and input devices (scale factor)
- Supports accesibility (ATSPI)
- Fully Themable (layout of the application included)
- Supports profiles
- Can take up as little as 8MB of space with a minimal set of dependencies
- Has a modular design

Why We Care About Optimization

- Moore's law doesn't apply to **battery** and memory bandwidth
- Anyways, we're no longer keeping up with Moore’s law!
- CPUs are gaining more and more blocks that are dedicated to a specific function, rather than gaining higher performance.
- Memory bandwidth directly limits most rendering operations.
- Many embedded devices have less available memory than a low end phone
  - Refrigerator, oven, dishwasher, washing machine, home automation…
- Even a low-end phone doesn't have much memory to spare once it runs a web browser!
- OpenGL consumes 10MB at best, usually more around 40MB. This is bad
- Multitasking!

Current State of Optimization

- Screen size is the primary driver of application runtime memory use.
- EFL/Enlightenment system has lower battery consumption than Android counter part
- EFL can fit in 8MB on disk (static compilation with minimal dependencies, including the Elementary widgets set)
- Minimal size for a single scenegraph and a minimal set of dependencies. Less than 1MB (Up to Ecore_Evas with jpeg/png, but no Edje support)
- No hard requirement on the GPU
- Enlightenment + Arch Linux combined:
  - 48 MB RAM
  - 300 Mhz (1024 x 768)
  - Yes, for a desktop profile!

Energy Efficiency

Energy Efficiency – Why?

- Get the most of a device's battery life
- Less heat to dissipate
- Allows for thinner devices
- Provides more freedom to designers
- Keep electronic device energy consumption at bay

Energy Efficiency – How?

- Optimize for speed.
- Optimize for memory.
- Optimize network use.
- Finally, optimize for battery!
- ...And maybe optimize the visual design too.

Optimize for Speed

- Obviously, it's more efficient to do only what is necessary!
- This applies to the GPU too
  - For user interface, this means supporting partial updates and reusing information from n image in the past to the next.
  - Trigger animations only at a speed the hardware is capable of displaying.
  - Don’t redraw the screen if nothing has changed!
- This is classic optimization work to catch the functions that are called and identify those that that should be called less.
- This is easy to do thanks to the debugging tools that are available (Don’t forget callgrind and kcachegrind).
- It's easy to write benchmarks and tests for limited hardware to see how things behave. Raspberry Pi devices are great for this purpose.

Optimize for Memory Use

- This also helps with energy efficiency!
- Accessing memory or disk is more costly than utilizing the CPU.
- Less memory accessed = less memory fetched = less energy consumed fetching it!
- Classic technical work to do here:
  - Improve cache locality
  - Linear access instead of random access
  - We saw a 5% increase in speed when deduplicating structure in memory with a kind of userspace copy on write (this mostly impacts the scenegraph walk logic).
- Always look for leaks and overuse with massif and massif-visualizer.

Optimize Network Use

- Today, most applications are connected; network use has an important cost.
- As more data is sent to the network, it's more likely that some bits will need to be retransmitted, using energy
- Only download what is needed, but there is a twist
  - Wireless stacks have multiple energy states, which require time to switch from one to another.
  - It's best to download all data in one go, then switch to full network idle for as long as possible. Otherwise, the system will stay in the high energy consumption state.
  - This is challenging because it requires a balance of probable future needs along with probable failure to download the right data.
- This is an unsolved problem. It requires synchronization of multiple tasks to download together, then idle at the same time to allow a system-wide low energy mode.

Optimize Battery Use

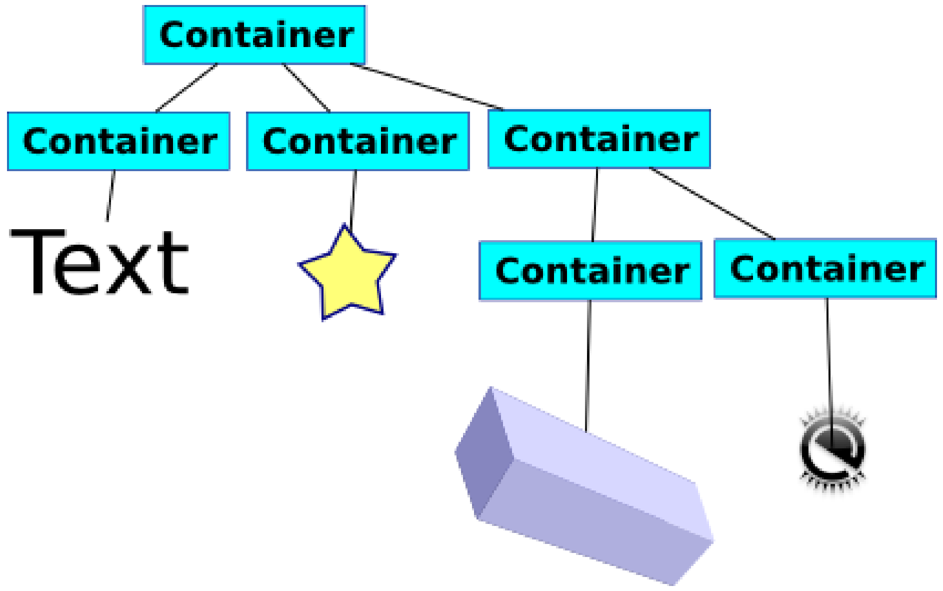
- This is not the only scheduling issue.
- The kernel has no clue what programs are about to do. Still, it tries to change the clock speed and number of active cores in a way it hopes will help.
- It has failed completely at this for years, leading to the creation of a userspace daemon that has a hard coded list of applications with their corresponding behavior.
- Where does the problem come from?
  - A scheduler that takes information from past process activity.
  - A scheduler that forgets everything as soon as it migrates to another CPU.
  - A CPU frequency driver that looks at the system load
  - A CPU idle driver that also looks also at the system load
- There is a fix coming: Energy Aware Scheduler.

Userspace and Scheduler

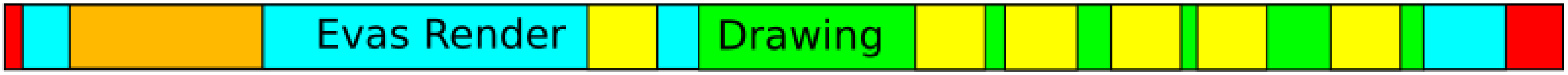
- This fix means:
  - Linking CPU frequency and idle to the scheduler
  - Keeping more information of past loads (in 20ms chunks)
- Interactive tasks change their behavior very quickly, and they can change their behavior during the rendering of frame (inside the 16ms of a frame).
- This doesn’t help the kernel figure out what applications will do in the future!

EFL Scenegraph

- Graph of primitive graphical objects
- Widget composed of primitives
- Doesn't rely on OS to draw widgets
- Compositor can also use an EFL Scenegraph
- Scenegraphs are a single place to optimize graphics calls by
  - Reusing data
  - Using cutouts
  - Limit shaders and texture switches
  - Split CPU, memory and IO bound processes



Efl - First Scenegraph



🟨 computing CPU intensive data

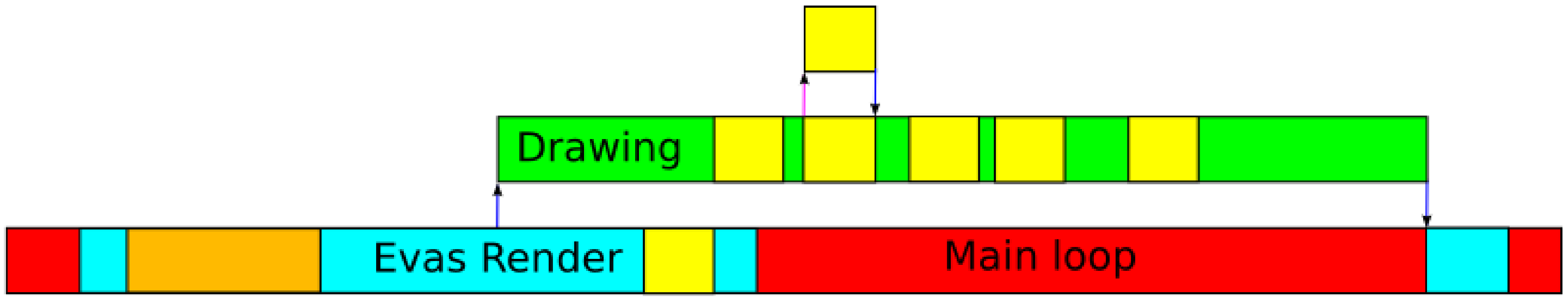
🟩 compositing data, mostly memory bound

🟧 layout object

🟦 walk tree to order operation

🟥 application logic

Efl – Scenegraph Today



🟨 computing CPU intensive data

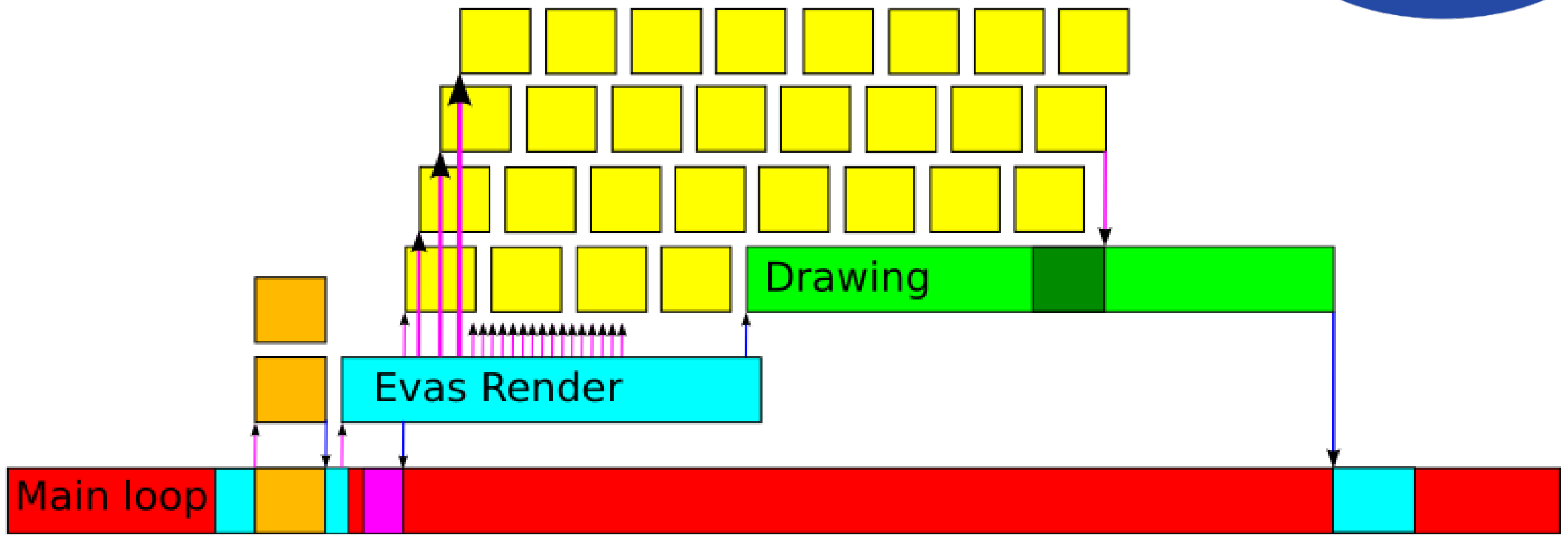
🟩 compositing data, mostly memory bound

🟧 layout object

🟦 walk tree to order operation

🟥 application logic

Efl – Scenegraph Tomorrow



🟨	computing CPU intensive data
🟩	compositing data, mostly memory bound
🟧	layout object
🟦	walk tree to order operation
🟥	application logic
🟪	waiting for COW

Userspace and Scheduler

- The main price to pay is an increase in threads
  - Overall memory consumption increases
  - Increase in complexity
  - More difficult to get right
  - More difficult to debug
- There's no way to explicitly tell the kernel what an application is about to do to provide hints about changes in processing needs when starting or stopping functions (this very unlikely to ever happen).
- Applications should also evolve this way, not just the toolkit.
- How can the toolkit better help applications?
- One interesting note: this pipeline change looks the same for both Vulkan and software rendering!

Optimize Visual Design

- This is something designers usually don’t want to hear!
- AMOLED screen do not consume energy to display black
  - For example, Reddit running in black consumes 41% less energy on Android!
- Full screen animations require a full screen update!
- Some screens have an integrated frame buffer that allows the CPU to completely stop everything once the screen is updated. Think smart watch!
- Finally, the more visual layers that are combined, the more complex a scene is and the more energy it needs
- Nothing beats a design based on a rectangle and vertical gradient (Just a series of optimized memsets).

Questions ?

Thank You!