

Optimizing key parts of the
DUNIA PIPELINE
Fast iteration for
FARCRY 4

Rémi QUENIN
Engine Architect - FarCry 4
Ubisoft Montréal
 @azagoth

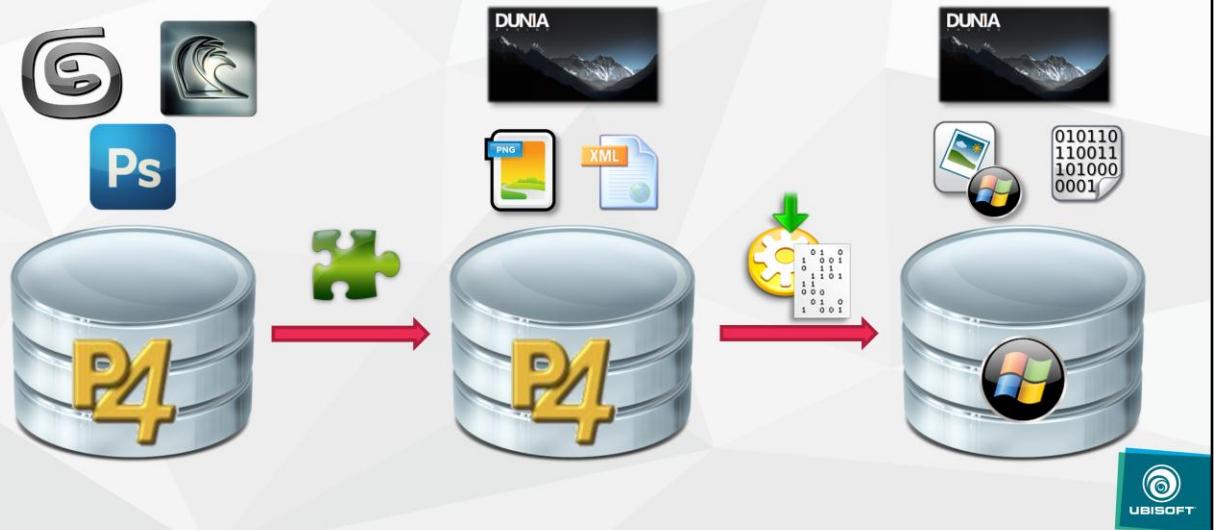


THE DUNIA PIPELINE

- 250 GiB of package per Day
Official + Test Maps
- Over 3.7 Million Lines Of Code
2.5 M. C++ Runtime
- 6 studios, 450 people in Montréal
Getting data everyday
- ~500 checkins per day
350 data, 150 code (+sound)



DATA FLOW IN THE DUNIA PIPELINE



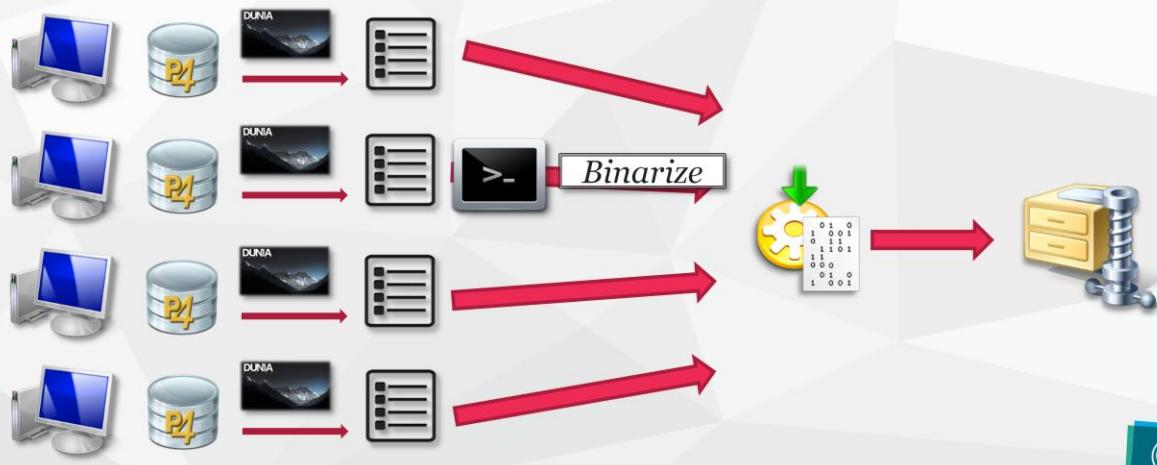
- 1 perforce instance for artists, to keep history of source asset files
- 1 other perforce instance, in which they export in a platform agnostic format, using plugins we provide
 - Design data also in this perforce instance (world description, entities, properties, ...)
- Can run the editor with a copy of this data
- JIT “compile” assets in order to use them, keep transformed asset on disk

DATA FLOW IN THE DUNIA PIPELINE



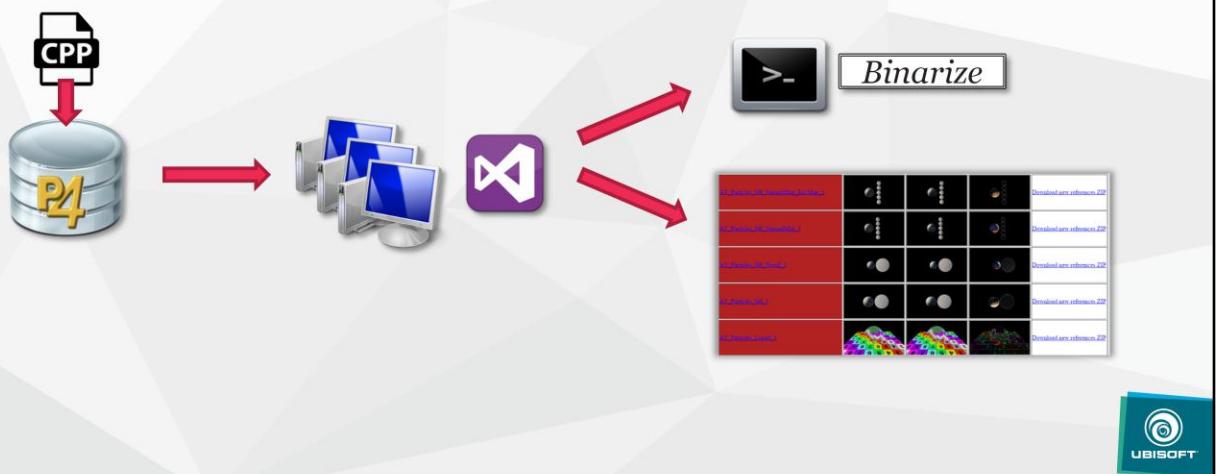
- To build a package for console
 - Extraction pass from the editor
 - Asset compilation pass (+ dependencies follow)
 - Compression + packaging
- Extraction can be long: 5~15 min per square kilometer

DATA FLOW IN THE DUNIA PIPELINE



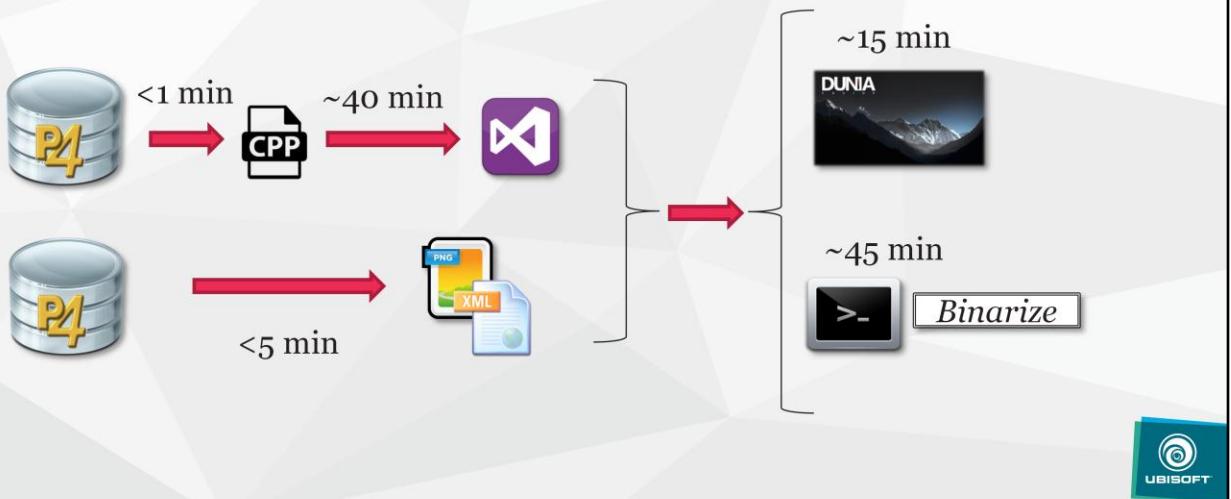
So distributed over night
Entire process known as “binarization”

AUTOMATED TESTS IN THE DUNIA PIPELINE



Binarization is tested with each submitted CL, alongside other regression tests

FC3 WORST CASE SCENARIO



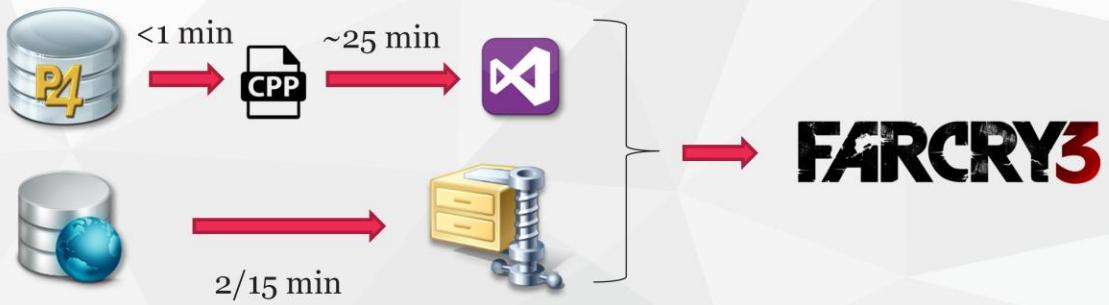
Full editor rebuild on FarCry3 : up to 40 min

First editor load up to 15 min because of JIT compiling

First binarization up to 45 min, because of extraction pass + first compilation



FC3 WORST CASE SCENARIO



To run on console, less code: 25 min full rebuild

+ copy latest nightly from network (2->15 min depending on network load)

Conclusion: optimization is mandatory for FC4 with 2 new platforms to support



OPTIMIZING *The Pipeline*

- 1 Optimizing Compilation Time
- 2 Optimizing Nightly Builds
- 3 Optimizing Package Synchronization
- 4 Optimizing Local Change Testing





1

From 40 min to 4 min



FASTBUILD

COMPILE TIME: FASTBUILD



HISTORY

- Editor DLL : ~40 min
3.7 Million LOC
- Unity/Blob builds : ~20 min
Bad Iteration
- Engine Architect pet project
FASTBuild by Franta Fulin
- Good State: All-in !
Early test show potential wins



FASTBuild

4
POINTS

 **PARALLELIZATION**

Maximize local resources

 **CACHING**

Share compilation results

 **BLOBBING**

Improve iteration

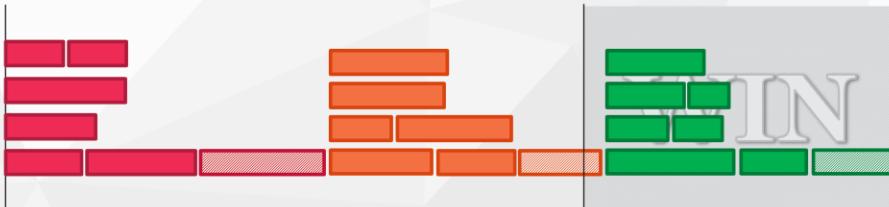
 **DISTRIBUTION**

Share HW resources



PARALLELIZATION

- Compiling/Linking : DLL



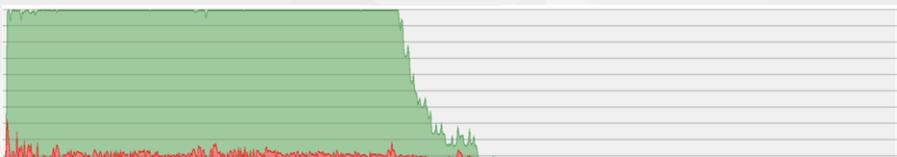
When compiling dependent DLL, only link steps depends on each other. Compilation can start as soon as possible

PARALLELIZATION

- Before



- After



MSBuild VS FastBuild

PARALLELIZATION

- Compiling : Static Libs



MSBuild also suffer from bad scheduling for static libs, where many projects can be started at the same time

PARALLELIZATION

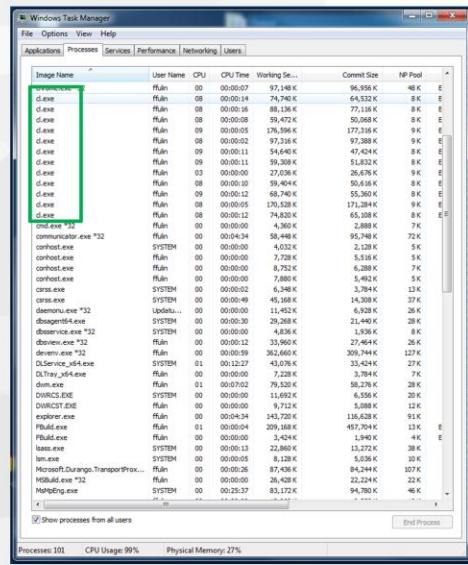
- Visual Studio:
 - Context Switching
 - File Cache eviction
- 32 Core Machine:
 - >1000 Processes!

Applications	User Name	CPU	CPU Time	Working Set	Commit Size	NP Pool
d3exe *32	RRun	00	00:00:00	6,244 K	4,120 K	12K
d3exe *32	RRun	04	00:00:08	129,321 K	138,388 K	14K
d3exe *32	RRun	01	00:00:10	141,544 K	145,152 K	17K
d3exe *32	RRun	02	00:00:07	40,344 K	45,088 K	13K
d3exe *32	RRun	06	00:00:07	31,460 K	21,480 K	13K
d3exe *32	RRun	01	00:00:09	32,044 K	144,584 K	14K
d3exe *32	RRun	00	00:00:09	4,836 K	2,536 K	8K
d3exe *32	RRun	03	00:00:05	137,164 K	148,824 K	15K
d3exe *32	RRun	03	00:00:08	134,712 K	144,632 K	15K
d3exe *32	RRun	03	00:00:08	128,321 K	128,388 K	14K
d3exe *32	RRun	04	00:00:09	133,400 K	145,760 K	15K
d3exe *32	RRun	01	00:00:09	124,276 K	128,924 K	16K
d3exe *32	RRun	02	00:00:04	39,408 K	36,364 K	13K
d3exe *32	RRun	03	00:00:04	39,408 K	145,084 K	17K
d3exe *32	RRun	04	00:00:04	40,176 K	37,118 K	13K
d3exe *32	RRun	00	00:00:09	6,456 K	4,844 K	17K
d3exe *32	RRun	03	00:00:09	138,156 K	147,364 K	15K
d3exe *32	RRun	02	00:00:08	128,321 K	128,388 K	14K
d3exe *32	RRun	03	00:00:08	128,321 K	138,232 K	14K
d3exe *32	RRun	01	00:00:12	125,516 K	122,700 K	16K
d3exe *32	RRun	02	00:00:09	139,438 K	151,072 K	16K
d3exe *32	RRun	03	00:00:09	128,321 K	146,584 K	15K
d3exe *32	RRun	00	00:00:00	4,844 K	2,544 K	8K
d3exe *32	RRun	00	00:00:00	5,028 K	2,536 K	8K
d3exe *32	RRun	03	00:00:12	146,416 K	151,200 K	15K
d3exe *32	RRun	04	00:00:09	135,444 K	148,804 K	17K
d3exe *32	RRun	03	00:00:09	135,544 K	147,638 K	15K
d3exe *32	RRun	03	00:00:09	27,624 K	23,988 K	13K
d3exe *32	RRun	02	00:00:14	139,724 K	116,740 K	15K
d3exe *32	RRun	03	00:00:14	139,724 K	123,272 K	14K
d3exe *32	RRun	00	00:00:00	5,048 K	2,648 K	8K
d3exe *32	RRun	07	00:00:05	34,200 K	21,132 K	13K
d3exe *32	RRun	06	00:00:09	38,468 K	25,732 K	14K
d3exe *32	RRun	03	00:00:02	27,624 K	23,988 K	13K
d3exe *32	RRun	02	00:00:02	36,264 K	31,294 K	13K
d3exe *32	RRun	00	00:00:00	5,966 K	3,728 K	8K
d3exe *32	RRun	01	00:00:00	25,588 K	22,656 K	13K
d3exe *32	RRun	01	00:00:05	62,060 K	39,236 K	14K
d3exe *32	RRun	02	00:00:05	46,256 K	46,280 K	12K



PARALLELIZATION

- FASTBuild:
 - NUM_PROCESSORS



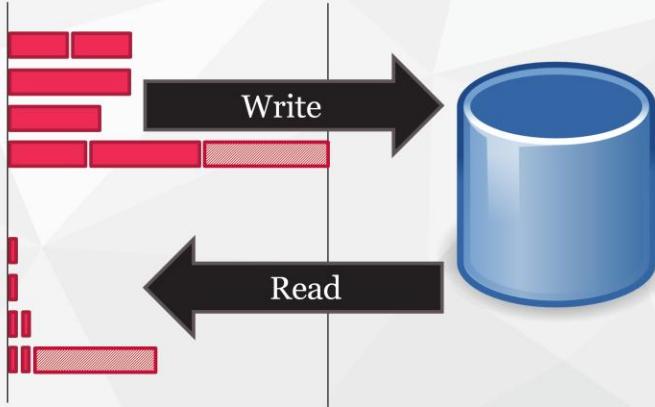
A screenshot of the Windows Task Manager showing a list of processes. The processes are listed in rows, with columns for Image Name, User Name, CPU, CPU Time, Working Set, Commit Size, and NP Pool. A green box highlights the first row for 'd.exe'. The 'NP Pool' column shows values like 8K, 16K, 32K, 64K, 128K, 256K, 512K, 1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M, 256M, 512M, and 1G. The 'Commit Size' column shows values ranging from 8.096 K to 94.768 K. The 'NP Pool' column shows values ranging from 8K to 1G. The 'Working Set' column shows values ranging from 7.148 K to 143.720 K. The 'CPU' column shows values ranging from 0.00 to 0.08. The 'CPU Time' column shows values ranging from 00:00:07 to 00:25:37.

Image Name	User Name	CPU	CPU Time	Working Set	Commit Size	NP Pool
d.exe	SYSTEM	0.00	00:00:07	7.148 K	8.096 K	8K
d.exe	SYSTEM	0.00	00:00:14	74.740 K	64.532 K	8K
d.exe	SYSTEM	0.00	00:00:16	88.196 K	77.118 K	8K
d.exe	SYSTEM	0.00	00:00:08	59.472 K	50.068 K	8K
d.exe	SYSTEM	0.00	00:00:09	78.368 K	173.934 K	8K
d.exe	SYSTEM	0.00	00:00:02	97.236 K	97.388 K	9K
d.exe	SYSTEM	0.00	00:00:11	54.640 K	47.424 K	8K
d.exe	SYSTEM	0.00	00:00:11	59.308 K	51.832 K	8K
d.exe	SYSTEM	0.00	00:00:03	77.048 K	26.856 K	8K
d.exe	SYSTEM	0.00	00:00:10	59.404 K	50.696 K	8K
d.exe	SYSTEM	0.00	00:00:12	68.740 K	55.360 K	8K
d.exe	SYSTEM	0.00	00:00:05	170.538 K	171.288 K	9K
d.exe	SYSTEM	0.00	00:00:00	74.424 K	62.584 K	8K
g.exe	SYSTEM	0.00	00:00:00	4.360 K	2.888 K	7K
g.exe	SYSTEM	0.00	00:04:34	58.448 K	95.744 K	72K
comhost.exe	SYSTEM	0.00	00:00:00	4,032 K	1,232 K	5K
comhost.exe	SYSTEM	0.00	00:00:00	7,700 K	5,332 K	5K
comhost.exe	SYSTEM	0.00	00:00:00	8,752 K	5,288 K	5K
comhost.exe	SYSTEM	0.00	00:00:00	7,880 K	5,400 K	5K
cras.exe	SYSTEM	0.00	00:00:02	6,348 K	5,784 K	13K
cras.exe	SYSTEM	0.00	00:00:49	45,384 K	14,936 K	37K
deservice.exe	SYSTEM	0.00	00:00:00	114.000 K	6,832 K	2K
deservice.exe	SYSTEM	0.00	00:00:30	29,268 K	21,440 K	28K
deservice.exe	SYSTEM	0.00	00:00:00	4,836 K	1,936 K	8K
desview.exe	SYSTEM	0.00	00:00:12	33,960 K	27,464 K	28K
desview.exe	SYSTEM	0.00	00:00:00	38,864 K	30,392 K	27K
DLSvcService.exe	SYSTEM	0.01	00:13:27	43,076 K	32,456 K	27K
DLTray_x64.exe	SYSTEM	0.00	00:00:00	7,238 K	3,784 K	7K
dm.exe	SYSTEM	0.01	00:07:02	79,520 K	58,276 K	28K
dm.exe	SYSTEM	0.00	00:00:00	124,000 K	65,684 K	28K
DIRECT2D_E	SYSTEM	0.00	00:00:00	9,712 K	5,096 K	12K
explorer.exe	SYSTEM	0.00	00:04:34	143,720 K	116,628 K	91K
FBuild.exe	SYSTEM	0.01	00:00:04	209,588 K	457,704 K	13K
FBuild.exe	SYSTEM	0.01	00:00:00	14,968 K	14,968 K	4K
Ime.exe	SYSTEM	0.00	00:00:13	22,860 K	13,272 K	38K
Ime.exe	SYSTEM	0.00	00:00:05	8,128 K	5,032 K	10K
Microsoft.Durango.Transport.Pox...	SYSTEM	0.00	00:00:26	87,496 K	84,244 K	107K
MBuild.exe	SYSTEM	0.00	00:00:00	26,428 K	22,224 K	22K
MBuildEng.exe	SYSTEM	0.00	00:25:37	83,320 K	94,768 K	4K



Improved utilization, what next? Eliminate redundant compilation : cache

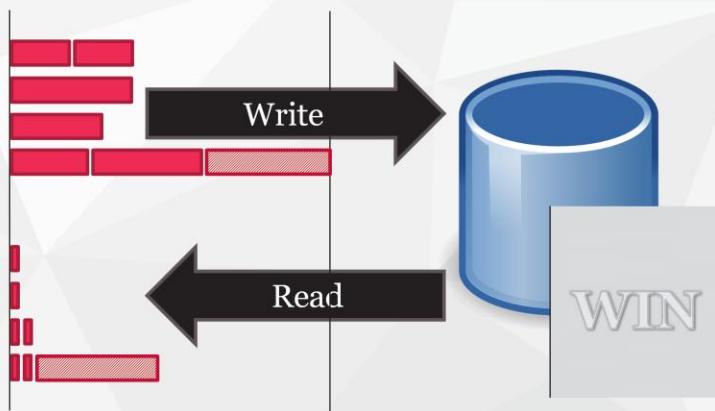
CACHING



Stores object files onto central cache, then link locally

Other user can retrieve object files directly out of the cache, then link locally

CACHING



CACHING

- Before



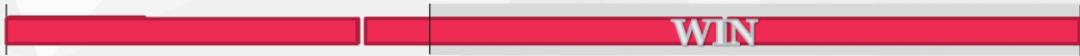
- After



MSBuild VS FASTBuild with cache

BLOBBING

- Reducing header compilation



Blobbing reduce a lot compilation time by reducing header compilation

BLOBBING

- Iteration



Problem: when you iterate, entire blob needs recompilation

Solution: extract edited file from the blob, while maintaining other blobs stable

DISTRIBUTION



WIN

Cut compilation in 2 separate steps:

- preprocessing, then compilation
- Preprocessed files have no file dependencies, and can be sent for remote compilation

MISCELLANEOUS

- Link Dependencies
Prevent useless relink
- Batching
Build machine, pre-submit checks



REAL TIMINGS

- Editor-X64-Release DLL (before)
 ~40 mins
- Editor-X64-Release DLL BLOB (before)
 ~20 mins
- Incredibuild
 ~12 mins
- FASTBuild
 ~10 mins (local PC!)
 <4 mins (cache/distribution)
 PS4: <1 min (cache/distribution)



EDIT&CONTINUE

- Too Much Code: Takes minutes
+Microsoft dropping support
- DLL with Hot Reload
Another story: lot of great content available online
- Incremental Linking
Where it actually helps



Now using hot reloadable DLL instead of edit and continue



2

4 Times Faster

NIGHTLY **BUILDS**



Improvements

FARCRY3



1.5 hours

75 Worlds

FARCRY4



HOW ?



Understand, optimize



Share



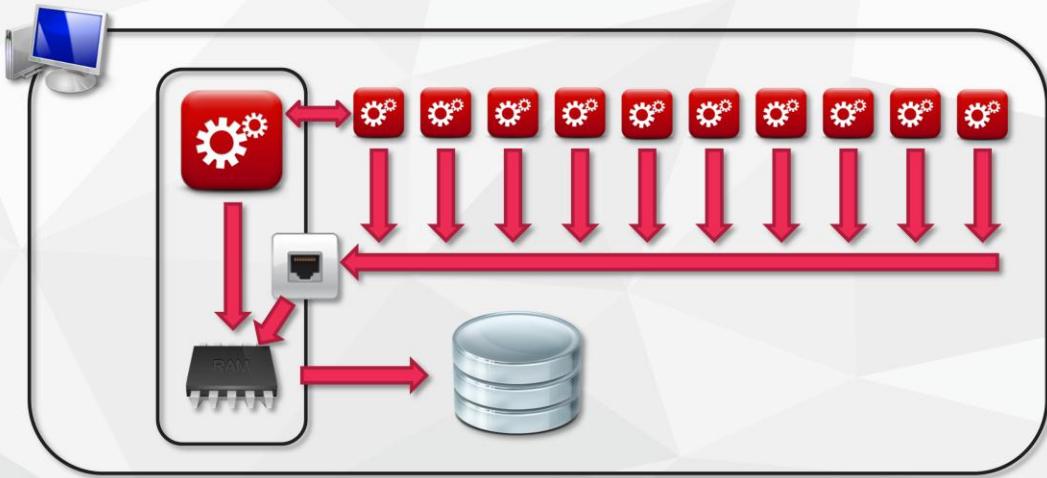
Profiling

- Multi-Process
Process isolation
- Multi-Machine
Cluster of Workers
- Interactions
Working together
- Big Picture
Understand



Assets get compiled in process isolation: hard to correctly sees what are the interactions between the main process and worker processes

Remote Profiling

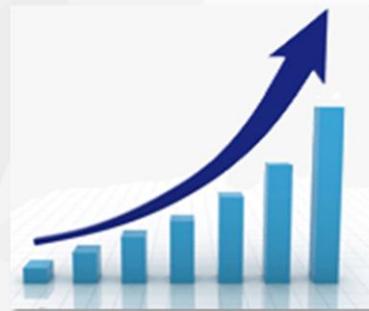


Remote profiling:

- sub process report profiling data over the network
- Data get committed to main profiling buffer

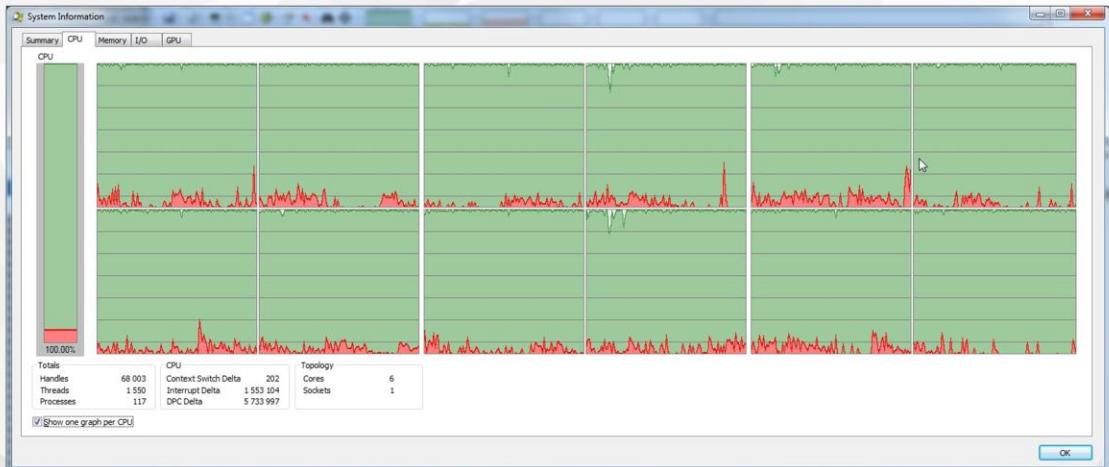
Remote Profiling

- On Top of Regular Tool
Custom Performance Analyzer
- Processes Shown as Threads
Analyze Interactions
- Fix !
Address Inefficiencies

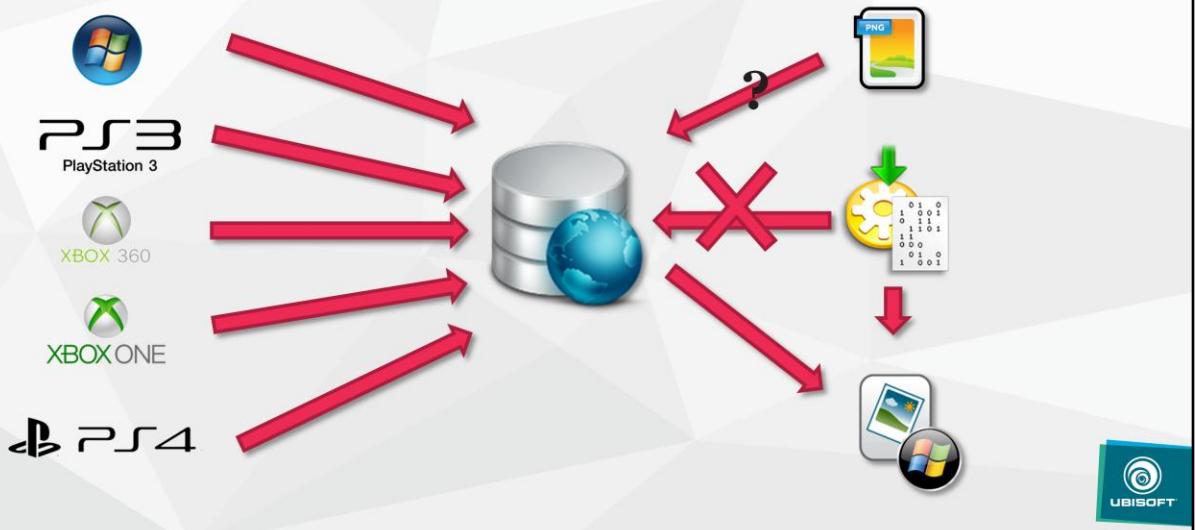


Remove useless steps
Removes unnecessary sync points
Better scheduling

Improvements: CPU Usage



Resource Caching



Store transformed asset on a central cache, so it can be retrieved by other users

Resource Caching

- Share
Prevent useless work
- Build Machine
Populate at night
- Editor
JIT Compilation (Just In Time)



Editor benefit from the cache since it JIT compiles asset

NIGHTLIES



Complex System

Key: Understanding



Prevent useless work

Hardware is crucial



FUTURE WORK

→ **Prevent Editor Usage**
Current Bottleneck

→ **More Incremental**
Some parts are not





3

From 20min to 1sec



OPTIMIZE PACKAGE SYNC.





2012 SUMMER

FARCRY3

FARCRY3

AC3

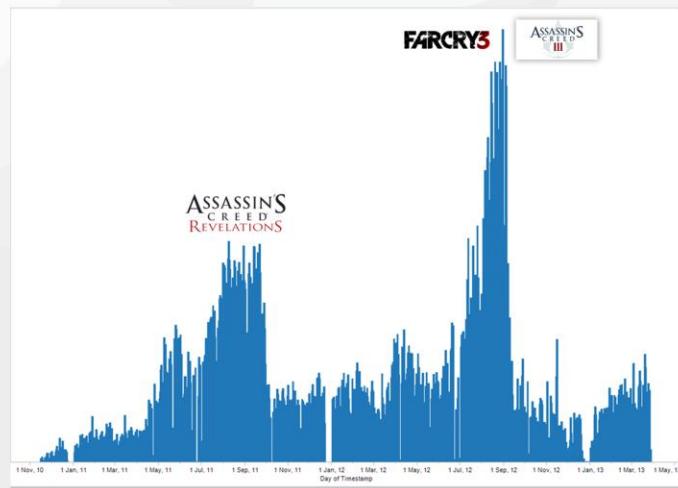


Network



2 major AAA games shipping at the same time
It killed our network

NETWORK



3

OPTIMIZE PACKAGE SYNC.

3.1

SOFTWARE: RTPAL



3.2

HARDWARE: ASSET STORE



Problem needed to be solved at the project level (software), as well at the studio level (hardware)



3.1

Software

SYNCH. PACKAGE: RTPAL



We'll get to what exactly is RTPal, first get back to summer 2012

PC BUILD

- Differential Uploads
- Sliced ISO
- ~50% Saving

Upload

STEAM PIPE



...shipping our PC build, using “steam pipe” form valve

We have an advantage over steam: we know our data

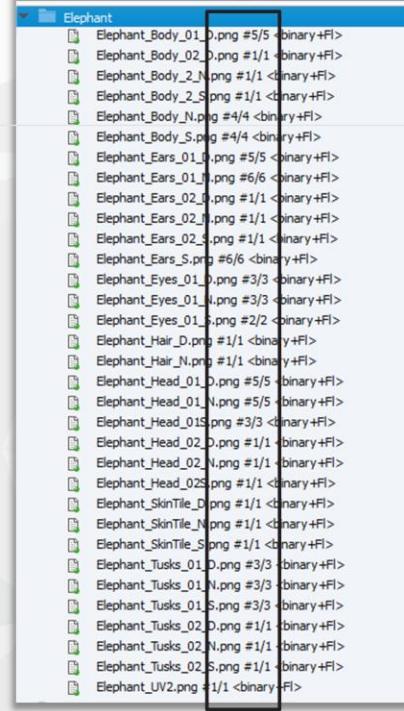
WE KNOW OUR DATA

- Big Files
File boundaries
- Redundancy
Same Resource, Several Worlds
- Amount of new Data ?
Out of 20GB package



WE KNOW OUR DATA

- Textures: Main Resource
80% of a package
- Textures: Revisions
Average 3 revisions



SOLUTION

```
<File  
    Name="common.dat" ←  
    Hash="12390181022458953888"  
    Size="146445768"  
    CreationTime="130360306344898459"  
    LastWriteTime="130674356947830393"  
    LastAccessTime="130674356947830393"  
    Attributes="1">  
  
<Part ←  
    Begin="0"  
    Size="49516"  
    MD5="02ae4cbce570fc9cdc8d656c19776036" />  
<Part ←  
    Begin="49516"  
    Size="4285"  
    MD5="6aa518d432ae603ec8e49497faca5c4" />  
<Part ←  
    Begin="53801"  
    Size="7691"  
    MD5="ef848a0c71b0dc64f62c838e22b2cb95" />  
<Part ←  
    Begin="61492"  
    Size="3118"  
    MD5="042ec74c70d3b5cb11b984553f27e19f" />  
<Part ←  
    Begin="64610"  
    Size="7691"  
    MD5="ef848a0c71b0dc64f62c838e22b2cb95" />  
</File>
```

Package
SLICING
Into Parts



SOLUTION



LINK



SOLUTION



VERSION 1



VERSION 2



RESULTS

- 1GB new per 22GB package
- 40% Shrink Within Package

Reuse

95%

2 Versions



RESULTS

- FC3: Few weeks + Milestones
- FC4: >10 000 Manifests

History

1 YEAR

Full Prod. Proof



THIS IS GOOD, BUT...

- Regular Workflow...
Gym/Test Map
- 20% of the package
 $20\% \text{ of } 5\% = 1\%$



THIS IS GOOD, BUT...

- Linking...
20 GB to write
- Deploy...
zzzzz.....



.....ON DEMAND.....

- **STREAM** parts on demand
- **OPEN WORLD** game
- **NETWORK** is faster than crappy **HDD**



Streaming the parts on demand is not a problem, as we are making an open world game that already streams its content asynchronously

FILE SYSTEM

- **VIRTUALIZE** File Access
- Regular **DISK**
- **MANIFEST** Content



Requires in-game code changes to stream parts: done by virtualizing file accesses

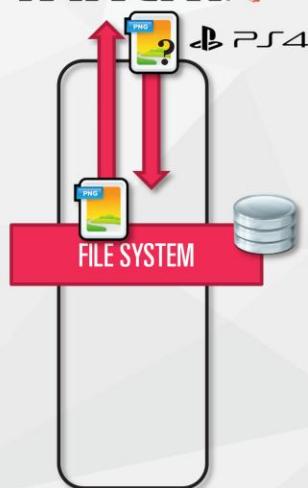
Several implementation then:

- Regular disk
- Virtualize manifest content
- Network file system
- ...etc

File systems can be combined : this is the file system stack

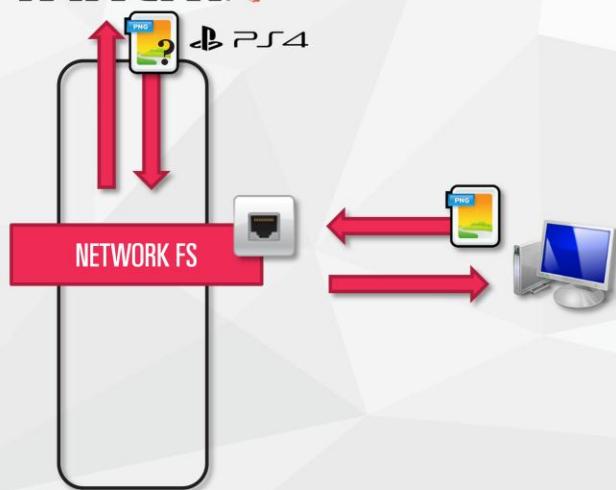
FILE SYSTEM STACK

FARCRY4



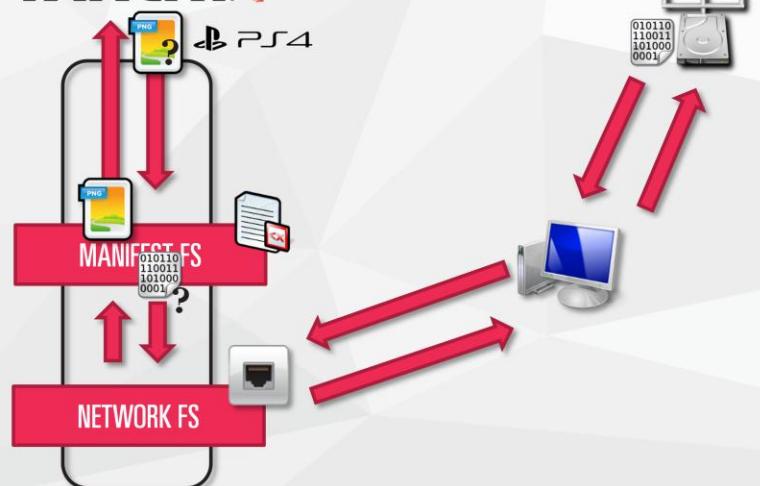
FILE SYSTEM STACK

FARCRY4



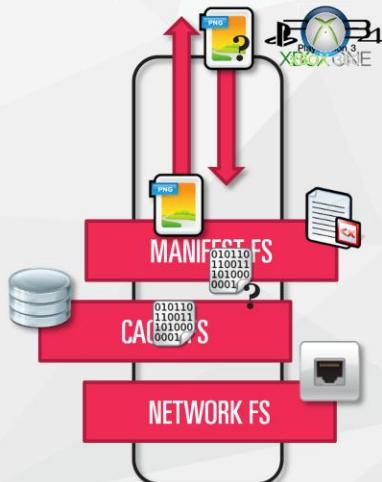
FILE SYSTEM STACK

FARCRY4



FILE SYSTEM STACK

FARCRY4



Network file system requires a companion app on the PC: RTPal

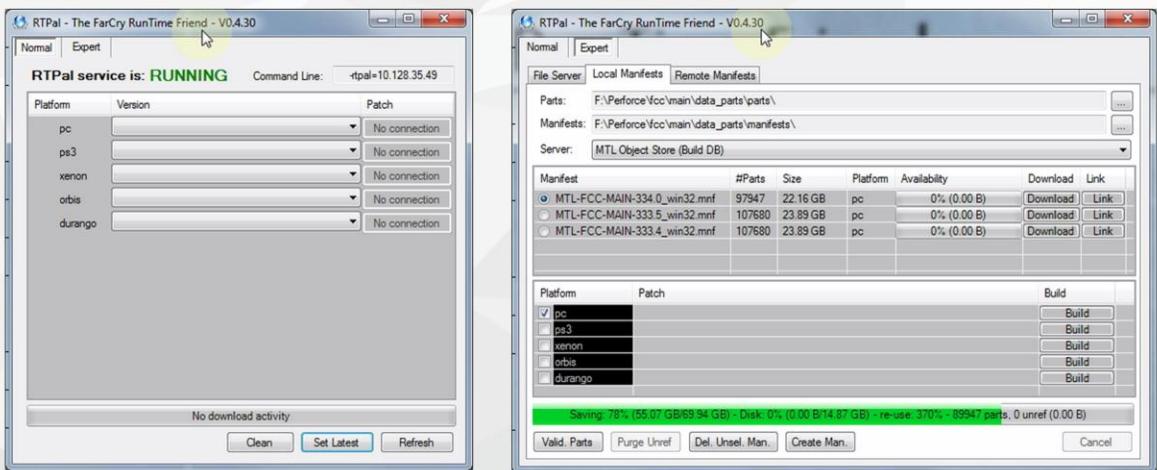
Workflow described is platform agnostic: unifies the workflow for all platforms

RTPal:

- Runs its own file system cache
- Retrieve manifests
- Retrieve and deliver parts on demand
- Do part caching

To run a package, select it in rtpal, and set the “–rtpal=ip” command line

RTPAL



1st video: sync 5 packages in a clic

2nd video: change PC package by just selecting it

3rd video: downloading some parts, demonstrate sharing in action - By getting parts for one package, other package are also progressing since they are referencing the same parts

RTPAL

STORAGE AND TRANSFER REDUCTION

History, Transfers

UNIFIED WORKFLOW

For all platforms

INSTANT PACKAGE SYNC

1 click



RTPAL

Re-Invents Package Distribution

By Getting RID of Package Distribution



END: 38 min



3.2

Hardware

SYNCH. PACKAGE: ASSET STORE



NEEDS

 **HIGH PERFORMANCE**
IOPS/Throughput

 **QUICKLY SCALABLE, HIGH CAPACITY**
Adapt to workload

 **ROBUST**
Failure tolerance



2012 SOLUTIONS



- CEPH: not good enough for IOPS in 2012
- HADOOP HDFS: centralized server
- OpenAFS: not mature enough
- => redhat implementation of the Gluster FS

GLUSTER FS

- Distributed file system
Cluster of PCs
- Replication between nodes
Up to 3 copies of a file
- 2 Network interfaces
One for internal data exchange



GLUSTER FS

- 8 x High End PCs
HP Z420 (32GB RAM, 6 HT CPUs 3.2GHz)
- 8 x High End SDD
On SATA3 RAID0 Controller card
- 2 x Network Interfaces
10 Gb/s



RAID Controller = LSI MegaRAID 9260-8i (PCIe 2.0, 8 lanes)

TEST GLUSTER FS – 8 Nodes

THROUGHPUT – 30 users, 14 GiB

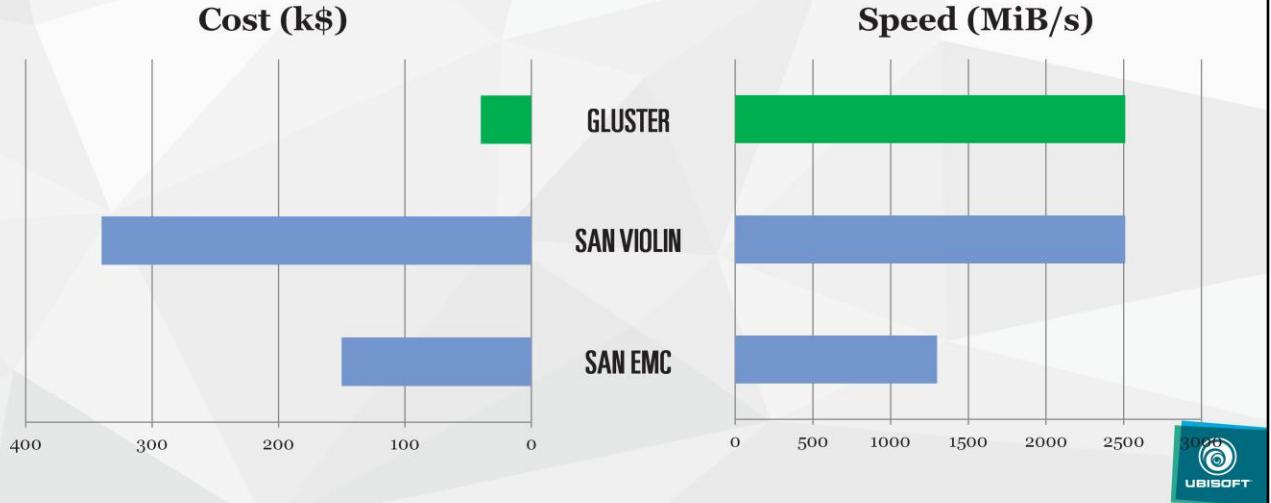


IOPS – 100k x 1KiB



SAN EMC VNX on NAS windows server

GLUSTER - COST



SAN VIOLIN : 6600 series on NAS Windows Server

SAN EMC : VNX on NAS Windows Server

Violin (and gluster) could probably go faster, but was limited by our testing infrastructure

USAGE SCHEMES

- RTPal
Package distribution
- Resources caching
Transformed assets
- Code compilation cache
FASTBuild



FUTURE WORK



Re-Evaluate CEPH

Improved since 2012



Improve Healing Process

If still with Gluster





4

From a Day to Minutes

LOCAL ITERATION: DEVPATCHER



2012 SUMMER

FARCRY3

FARCRY3

Fixes



Nightmare



While shipping FC3, testing small changes, 3 solutions:

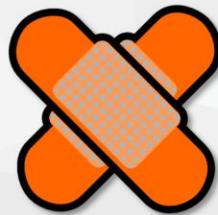
- Change, submit, wait next day for nightly – not verry good workflow...
- Change, binarize locally – too long...
- Change, open bigfile by hand, locate and replace file by hand, save, deploy

3rd workflow was manual, but actually efficient. Could we automate it ?

WHAT IF...

- Detect **CHANGES**
- Handles **COMPILE**
- Handles **BIG FILE** creation

Introducing
PATCHING
as a Workflow



CREATE PATCH

3
STAGES

→ DETECTING CHANGE

Compile Dep.

→ PACKAGING CHANGE

Patch BigFile

→ APPLYING CHANGE

BigFile Stack



The “DevPatcher” is the tool that automates all those steps

1 DETECTING CHANGES



Understand what are all the real inputs that defines an output

Write all of those input down into a description file (CRC of files, code version, parameters, ...etc)

=> This description file is called a “Compile Dep”

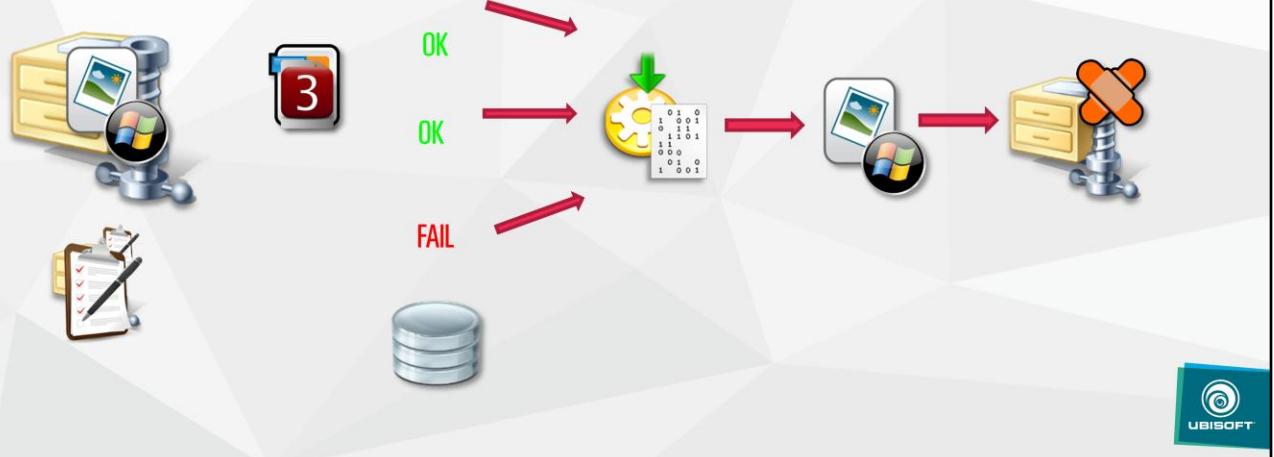
2 PACKAGING CHANGES

NIGHTLY BUILD



All “Compile Deps” gets packaged alongside the actual data with each nightly

2 PACKAGING CHANGES



To know if an asset has been changed locally compared to the official build, don't get the official build: just get compile deps

- Examine all inputs, and compare to local disk
- If there is a change, asset needs to be rebuild
- All re-built assets are packaged together in a patch bigfile

3 APPLYING CHANGES

FARCRY4 PRO ?



- Patch bigfile gets mounted with precedence over regular package
- Get rid of (or just unmount) the patch bigfile to get back to the official build

DEVPATCHER & RTPAL

RTPal



Compile Dep.



STACK FS



- Tx to rtpal, nothing gets downloaded
 - Just the compile deps -> patch BF
 - If asset in patch bf, get it, else stack fs -> download
- => don't have to download an asset to patch it: INSANE !



DUNIA *The Pipeline*

- 1 FASTBuild
- 2 Fast Nightly Builds
- 3 RTPal & Asset Store
- 4 DevPatcher





THANKS & CREDITS

- Franta Fulin / *FASTBuild* (<http://fastbuild.org/>)
- Jean-Francois Cyr / *DevPatcher*
- Olivier Deschamps / *DevPatcher*
- Laurent Chouinard / *AssetStore*
- Jonhatan Chin / *AssetStore*
- Jocelyn Hotte / *AssetStore*





Rémi QUENIN

 @azagoth
remi.quenin@ubisoft.com

