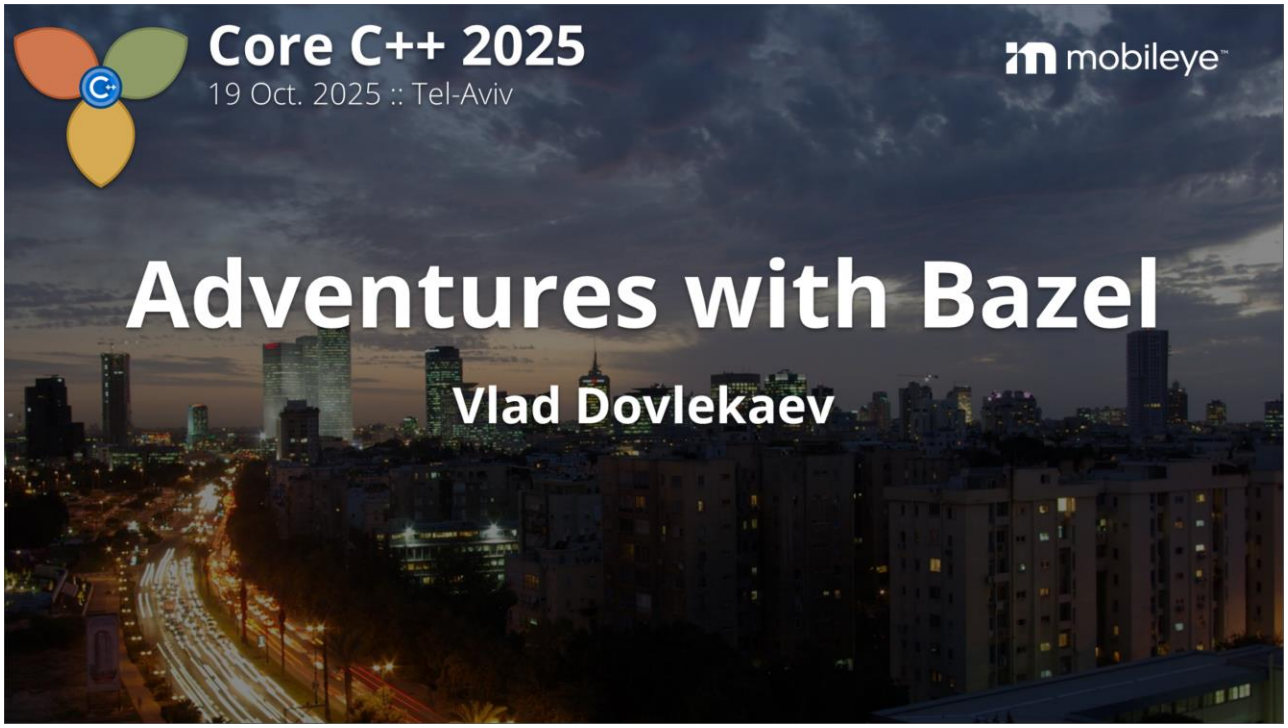

Core C++ 2025
19 Oct. 2025 :: Tel-Aviv
mobileye
Adventures with Bazel
Vlad Dovlekaev

Agenda

- Little bit about us
- What's Bazel and why would you care?
- Bazel @ Mobileye
- What it takes to transition & maintain

1 Little bit about us

Mobileye legacy codebase

- Multi-platform embedded project: x86, MIPS, RISCV, custom SoC
- Large scale: >200 repositories, >20 MLOC
- Multi-language: C/C++, Python
- Large portion of old legacy code, 20+ years
- Open-source contributor: LLVM, Linux, …
- Mixed legacy build systems: make, cmake, waf
- Build time: let's not even talk about it.

Transition to the new build system is not like re-writing some old legacy component – it's more like archeological discovery expedition.

The old build system is not just an old component – but rather a knowledge base accumulated many year of various people ideas as to how to make this system work in various scenarios – so the purpose of the build system transition is more of excavating all this knowledge and basically re-design it from scratch

1. There are a lot of special/unusual use cases and plumbing that only few original authors knows the purpose of
2. The number of "temporary" workarounds and "sure will fix it later" cases and other "todo" is huge
3. The amount of dead code that nobody is dare to remove (see the 1 item)

Pros: you will know a lot of new people – who where the original code owners then and could be big bosses now ☺

2 What is Bazel

What is Bazel

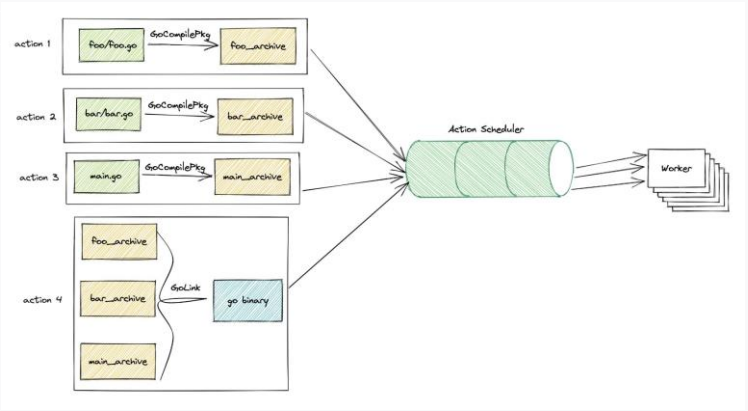
- Open-source build system developed by Google
- Designed for large, complex software projects
- Multi-platform, multi-language support
- Python-based build language
- Built-in support for Remote Build Execution (RBE)

Used by: ASML, BMW, LinkedIn, Lucid, Lyft, Nvidia, Pinterest, Tinder, Twitter (migrating from Pants to Bazel), Uber, VMWare, Wix,
Open Source: Angular, Gerrit, Kubernetes, Selenium, Spotify, TensorFlow,

What is Bazel

Remote Build Execution API (REAPI)

- Open standard for remote action execution
- Bazel only defines the API
- Third-party implementations



Idea:

- decomposite the large target into small pieces – actions (Ex. single C++compilation unit) and distribute to many remote executors
- for each small action provide access to inputs, tools, command line, environment if needed, and gather outputs

Questions:

- how to provide inputs ? shared storage, direct copy ? **A: copy**
- should we always provide all the inputs or give the tool the opportunity to detect? A: **no detection**, not all tools can detect. Should be **fully and statically provided** before running an action. Enforced by **sandboxing**
- should we always specify the expected output ? A: yep, built-in rules supply it under the hood, custom rule should **supply explicitly**

What is Bazel

Dependencies definitions

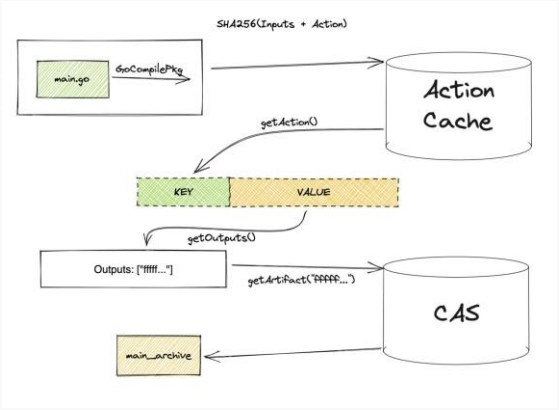
- full and explicit static declaration of dependencies
- missing dependency detection through sandboxing

```
cc_library(
    name = "debug_client",
    srcs = [
        "ipc/SocketClient.cpp",
        "ipc/SocketClient.h",
        "ipc/DebugClient.cpp",
    ],
    hdrs = ["ipc/DebugClient.h"],
    includes = ["ipc"],
    deps = ["@com_google_protobuf//:protobuf"],
    linkstatic = True,
    target_compatible_with = ["@platforms//cpu:x86_64"],
    visibility = ["//visibility:public"]
)
```

What is Bazel

Caching and incremental builds

- content hash to detect changes
- CAS and Action Cache
- deterministic actions
- always incremental build
- caching the test results?
- early cut-off



Questions:

- Should we copy the entire set of dependencies each time (even if the action was run before) ? **A: no, inputs are cached on the server side**
- How it is stored, by name, path, timestamp? **A:** using content hash - **introducing CAS storage**
- What about caching the output (it could be used as an input or even tool !!! to further actions)? **A: cache it as well**
- Is it always safe to cache the output artifacts? **A:** we assume **predictable & reproducible actions (bit exact)**.
- What about the compilers, achievers, linkers:

  * compilers are not reproducible by default (timestamps, absolute paths in debug info) **A:** could be configured to behave deterministically: (`gcc/clang: -frandom-seed, -fdebug-prefix-map. msvc: /Brepro`)

  * linker are not reproducible by default (timestamps, non-stable symbol ordering, section ordering, build id). **A:** (`-frandom-seed, --build-id`)
- What if the tool doesn't produce any artifact but only output stream? (example: failed compilation) **A:** cache it as well – **introducing AC**

Implications and side effects:

- it doesn't matter where the action is executed – enough to be executed once in the enterprise, everybody else will get the output of the same action from cache
- almost all the builds are just incremental builds

Questions:

- if the output artifact could be used as tool for further actions, can we unit-tests as just another build target and then run it? **A:** yep, if tests are reproducible/stable
- What about higher-level tests? **A:** yep, with some effort (software defined infrastructure if applicable)
- What if the change in the input doesn't cause the change in the output ? **A:** mention the early-cut off (next slide)

What is Bazel. Early Cut-off

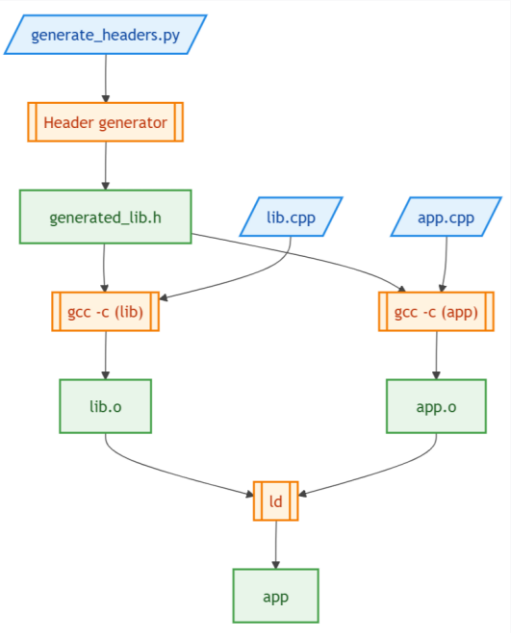
Avoid unnecessary rebuilds when changes are made

Scenario 1

cosmetic change in `generate_headers.py`

Scenario 2

functional change in `generate_headers.py` causing cosmetic change in `generated_lib.h`



What is Bazel

External projects integration

- no need to keep external repos in SCM
- hermetic dependencies

```
local_repository(
    name = "umd",
    path = WORKSPACE + "proprietary/umd",
)

http_archive(
    name = "gbenchmark",
    sha256 ="bdefa4b03c32d1a27bd50e37ca466d8127c1688d834800c38f3c587a396188ee",
    strip_prefix = "benchmark-1.5.3",
    urls = ["https://github.com/google/benchmark/archive/v1.5.3.zip"],
)
```


Questions:
what if my target depends on external code? A: should be also **hermetically defined.**

open source (3 – options, create bazel build files, build as legacy, use pre-build binaries)

What is Bazel

Multiple language support

- built in support for C/C++, Python, Java, Go, ...
- extensions for almost any other language
- generic rules for simple cmd-based actions
- custom rules for anything else
- macros
- multi-language dependencies

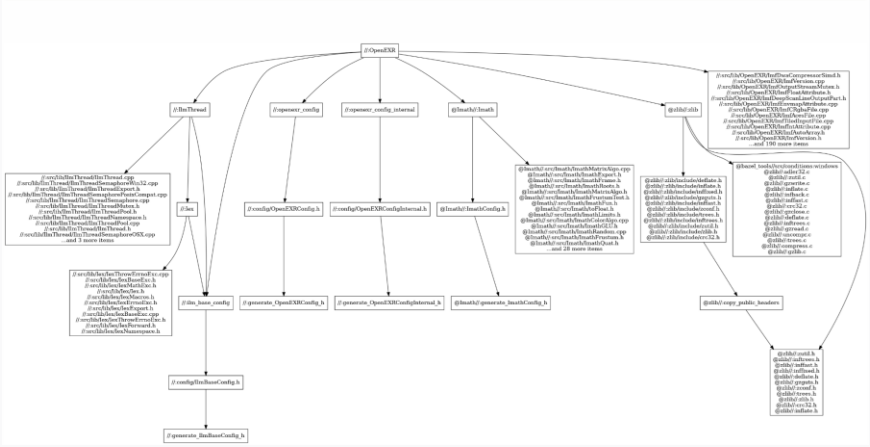
```
cc_binary(
    name = "phdl.cpython-37m-x86_64-linux-gnu.so",
    srcs = ["common/simpleDriverPythonModule.cpp"],
    deps = [
        "@python37//:python37",
        "//buildenv/technology/common:headers",
        ":phdl_solib"
    ],
    linkshared = True,
    visibility = ["//VMP:__subpackages__"],
)

py_library(
    name = "phdl_py",
    data = [":phdl.cpython-37m-x86_64-linux-gnu.so"],
    imports = ["."],
    visibility = ["//VMP:__subpackages__"],
)
```

What is Bazel

Reflection and Queries

- query language
- aspect rules

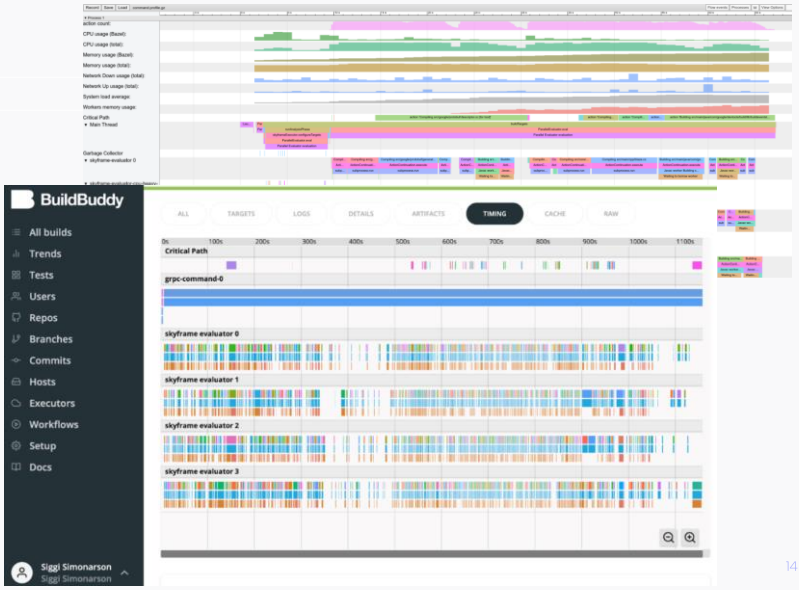
```
bazel query --noimplicit_deps 'deps(//:OpenEXR)' --output graph > graph.in
dot -Tpng < graph.in > graph.png
```



What is Bazel

Tracking and debugging

- Built-in Profiler
- Build Buddy



3

Bazel in Mobileye

Bazel in Mobileye - POC

- Get the feeling by building existing project (LLVM)
- Transition of isolated complex project
- Dedicated team for Bazel transition
- Bubble up manual transition

	CMake Build (full)	Bazel Build (full)	Bazel Build (avg)	Speed up (full)
LLVM (llc)	~10 min	~40 sec	~20 sec	15x
POC project	~7 min	~40 sec	~12 sec	10x

Build comparison: Dual Xeon server, 1.5 TB memory, 96 cores
Bazel build: remote execution, full cache.

Bazel in Mobileye – Scale Up

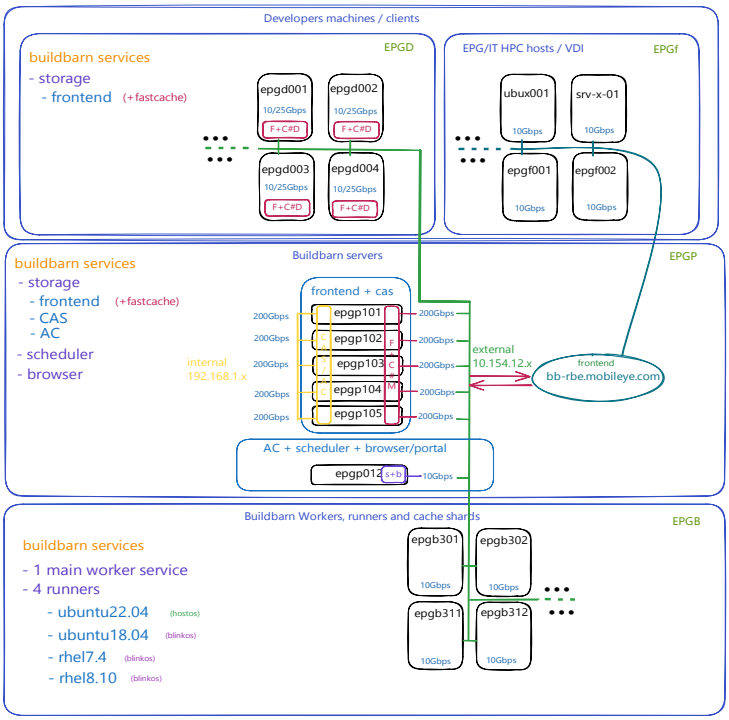
- Distributing transition effort vs Dedicated Team. Lesson Learned
- Developing automated translation tooling. Waf-to-Bazel
- Allows large project parallel branching transition
- Automatic generation of dependency specifications
- Frequent pull-downs from master branches

Bazel in Mobileye – Current status

- Transitioned code: ~10 MLOC
- Dev team: >500
- Build team: 3 dev + 1 IT support
- Build times improvements: ~10x

Bazel in Mobileye

- On-prem, based on BuildBarn services
- 600 TB CAS storage + 5 TB Action Cache
- Sharded across 5 nodes
- ~100 GB Key Location Map for CAS (index)
- ~10 GB KLM for AC (index)
- Load-balanced frontends with small local CAS
- >15 K workers cores



- Typical bottlenecks are network bandwidth between storage nodes and disk IO on block devices
- Additional bottleneck seems to be a buildbarn tool limitation, probably due to it being a cloud-driving tool it do not seems really optimized to run on 1 monster server
- Network bonding is used for high-throughput and redundancy on CAS storage nodes (2x100GbE for external traffic + 2x100GbE for internal traffic, replication)

3

Bazel - Summary

Bazel in Mobileye - POC Challenges

- Steep learning curve during the transition
- Remote execution infrastructure maintenance
- Network becomes a bottleneck
- External open source non-Bazel projects integration
- Handling of volatile information
- Post-build handling

Bazel in Mobileye – Surprises

- Bazel code maintenance is easy
- Handling C/C++ headers dependencies is not a big problem
- Low average load on remote workers

Bazel - Summary

Who is it for

- Large-scale projects which may afford a dedicated build team + IT

Not worth it if

- Small or POC projects
- Small to medium single-language projects with "dedicated" build systems (Rust with Cargo, Java with Maven/Gradle, Go with Mage, …)

Q&A

Backup

Ccache vs Bazel Cache

CCache:

- supports only C/C++ single file compilation caching (no link, no custom targets)
- some C/C++ compile time options are not supported (in direct mode)
- wraps the compiler executable, manipulating its options (compiler sensitive ?)
- sensitive to the workspace locations (requiring tuning of 'base_dir' for shared cache)
- require shared location for distributed scenario
- still rely on timestamps for include files (in addition to hash) – potential cache misses

Bazel Cache:

- supports arbitrary targets (multi language environment)
- not sensitive to workspace location (no need for NFS for distributed cache)
- strong cache correctness guarantee (action cache + CAS cache)
- built-in support for distributed builds for arbitrary targets