Breaking down the BitBake build on the process level

Amir Kirsh


Yocto Project Summit, 2022-05
About me

Developer Advocate at

Lecturer
Academic College of Tel-Aviv-Yaffo
Tel-Aviv University

Member of the Israeli ISO C++ NB

Co-Organizer of the CoreCpp conference and meetup group
EVERYTHING EVERYWHERE ALL AT ONCE

A FILM FROM DANIELS
First steps for a more efficient bitbake build
First steps for a more efficient bitbake build (1)

$ bitbake -k

-k, --continue  Continue as much as possible after an error. While the target that failed and anything depending on it cannot be built, as much as possible will be built before stopping.

First steps for a more efficient bitbake build (2)

sstate cache

https://wiki.yoctoproject.org/wiki/Enable_sstate_cache

First steps for a more efficient bitbake build (2)

sstate cache

https://wiki.yoctoproject.org/wiki/Enable_sstate_cache

I will talk more about caching in my talk tomorrow: "Augmenting sstate-cache with ccache"
First steps for a more efficient bitbake build (3)

Parallel build

local.conf in build/conf directory:

PARALLEL_MAKE
PARALLEL_MAKEINST

First steps for a more efficient bitbake build (3)

Parallel build

local.conf in build/conf directory:

- PARALLEL_MAKE
- PARALLEL_MAKEINST

By default, the OpenEmbedded build system automatically sets this variable to be equal to the number of cores the build system uses.

Parallel build

local.conf in build/conf directory:

```
PARALLEL_MAKE
PARALLELMAKEINST
```

By default, the OpenEmbedded build system automatically sets this variable to be equal to the number of cores the build system uses.


Setting different PARALLEL_MAKE value at the recipe level (+ reasons for doing that):
First steps for a more efficient bitbake build (4)

Additional advice by Yocto manual:

https://www.yoctoproject.org/docs/latest/dev-manual/dev-manual.html#speeding-up-a-build
Throwing in additional cores

- More powerful HW
- Distributing to additional machines (on-prem / cloud)
Throwing in additional cores

- More powerful HW
- Distributing to additional machines (on-prem / cloud)

But would it help?
Throwing in additional cores

- More powerful HW
- Distributing to additional machines (on-prem / cloud)

But would it help?

Where is the bottleneck?

Common HW resources a build consumes

- **CPU**
  - Multicore: if tasks exceed core number they will get queued
  - There are tasks that can utilize only a single CPU

- **IO**
  - Slow disks can cause waits until data is fetched or till a dependency file finished writing

- **Memory**
  - Swapping to virtual memory is possible but costly

- **Network**
  - Yocto has no separate “download” phase. Slow network would affect download speed and build time
Throwing in additional cores

network

cpu

Analyzing things on the process level

- Which tools are common across many recipes?
- Which tools take up most of the time?
- What are the bottlenecks on the process level?
Incredibuild View

distributable
not distributable (local only)
task which printed to stderr

Incredibuild View

- distributable
- not distributable (local only)
- task which printed to stderr

Data Analysis - CPU Wait

Number of minutes there are more tasks in queue than available local cores:

```sql
1 select
2 count(*)/60 as Number_Of_Minutes
3 from build_4_statistics
4 where DistributedWaitingTasks > 16
```

<table>
<thead>
<tr>
<th>Number_Of_Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
</tr>
</tbody>
</table>
Data Analysis - IO Wait
Data Analysis - IO Wait

```sql
select count(*)/60 as Number_Of_Minutes
from build_4_statistics
where IOWaitPercent > 1
```

<table>
<thead>
<tr>
<th>Number_Of_Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Data Analysis - Compiler Cache Potential

```sql
SELECT
    sum(processes.end - processes.start)/1000/60/16 AS total_duration_min_all_cores,
    round(avg(processes.end - processes.start))/1000 AS average_duration_sec,
    count(*) AS Count
FROM `build_4_process` processes
WHERE ($slotIsLocal==1) AND (ProcessName like "%g++")
ORDER BY
    count DESC
```

<table>
<thead>
<tr>
<th>total_duration_min_all_cores</th>
<th>average_duration_sec</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>2.6</td>
<td>17216</td>
</tr>
</tbody>
</table>
Top Time Takers Tasks:

- do_compile
- do_configure
- do_package (rpmbuild)
Parallelization - Special Challenges (1)

Agl 12.1 uses 321 different compilers

- Most of them are real files on disk (i.e. not a symlink)
- We see a few different gcc compilers and g++ compilers
Parallelization - Special Challenges (2)

Virtualizing the entire filesystem on the task level without overwhelming the network

- No need to maintain a homogenous environment on the other machines in the grid (no compilers needed on helper, no source – everything synced on demand)
- Task runs in total isolation from helper with full emulation of the filesystem of the initiator
Parallelization - Special Challenges (3)

Permission and user management required to make sure images created with IB are identical to stock Yocto builds, especially supporting pseudo and different forms of chroot
Parallelization - Benchmark Results

Benchmark
16 core AWS machines

<table>
<thead>
<tr>
<th></th>
<th>Without Incredibuild</th>
<th>With Incredibuild</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGL 12</td>
<td>190 mins</td>
<td>90 mins</td>
</tr>
<tr>
<td>Yocto 3.4</td>
<td>86 mins</td>
<td>49 mins</td>
</tr>
</tbody>
</table>

To Summarize

- We love Yocto and bitbake
To Summarize

- We love Yocto and bitbake
- The bitbake build can be parallelized to additional machines to gain faster build time

To Summarize

- We love Yocto and bitbake
- The bitbake build can be parallelized to additional machines to gain faster build time
- I will discuss caching in my talk tomorrow
Everything Everywhere All At Once
Breaking down the BitBake build on the process level


amir.kirsh@incredibuild.com