Speeding up development by setting up a kernel build farm

Willy Tarreau
HAProxy Technologies

Kernel Recipes 2016

Outline

- Target
- Context
- Solutions
- Pitfalls
- Improvements

Target

- Developers who build a lot of code
- Maintainers who backport lots of patches
- People who have to debug and bisect
- Developers having to use very slow laptops (“ultrabooks”)
- Those who like to have fun with clusters
- Others ?

Observations

- Backporting fixes into old kernels is not trivial
- It often causes build failures
- Need to build a lot to validate backports
- Build time dominates in a backport session
- Not always building from the same place
- I spend a lot of time building distros as well...

How is time spent

- Lots of time spent between keyboard and chair
- Few short build sessions (< 10s)
- Many medium build sessions (~ 1mn)
- Few long build sessions (>15mn)
- You never know how long it takes

The goal is in fact to reduce the **wait** time!

Ideas on how to reduce build time

- Stop testing backports and rely on previews :-)
- Release more often with less patches
- Buy a bigger machine
- Use a compiler cache
- Distribute the build over several machines

Ideas on how to reduce build time

- ~~Stop testing backports and rely on previews :-)~~
- Release more often with less patches
- Buy a bigger machine
- Use a compiler cache
- Distribute the build over several machines

Ideas on how to reduce build time

- ~~Stop testing backports and rely on previews :-)~~
- ~~Release more often with less patches~~
- Buy a bigger machine
- Use a compiler cache
- Distribute the build over several machines

Ideas on how to reduce build time

- ~~Stop testing backports and rely on previews :-)~~
- ~~Release more often with less patches~~
- ~~Buy a bigger machine~~
- Use a compiler cache
- Distribute the build over several machines

Ideas on how to reduce build time

- ~~Stop testing backports and rely on previews :-)~~
- ~~Release more often with less patches~~
- ~~Buy a bigger machine~~
- ~~Use a compiler cache~~
- Distribute the build over several machines

Ideas on how to reduce build time

- ~~Stop testing backports and rely on previews :-)~~
- ~~Release more often with less patches~~
- ~~Buy a bigger machine~~
- ~~Use a compiler cache~~
- **Distribute the build over several machines**

Distributing the build ?

For this, you need :

- A distributable workload (not kidding)
- Multiple machines
- The **exact** same compiler everywhere
- Some way to submit the job to these machines
- A low enough latency

Distributable workloads

Quite a few requirements :

- Strong support for parallel builds (make -j)
- No dependency on the build node...
- ...or ability to replicate the build environment
- Many more C files to build than machines
- Homogenous build times (hence sizes)

=> ***The kernel fits pretty well here.***

Distributing the build ?

For this, you need :

- A distributable workload (not kidding)
- Multiple machines
- The **exact** same compiler everywhere
- Some way to submit the job to these machines
- A low enough latency

Distributing the build ?

For this, you need :

- A distributable workload (not kidding)
- Multiple machines
-
- Some way to submit the job to these machines
- A low enough latency

Same compiler everywhere ?

It is absolutelely mandatory

=> **Use crosstool-ng for this even for the local node**

- Reusable, archivable configurations
- Builds relocatable toolchains that run everywhere
- Supports Canadian Cross compilers
- Supports bare-metal compilers
- Use of *arch-vendor-os-abi* naming allows many different toolchains to coexist

Hint: set the gcc version in the vendor field, eg x86_64-gcc47-linux-gnu

Distributing the build ?

For this, you need :

- A distributable workload (not kidding)
- Multiple machines
- The **exact** same compiler everywhere
-
- A low enough latency

Distributed build controller

This component will be responsible for distributing the load across all the machines. There are a few prerequisites :

- Must not be intrusive in the build process :
  - No extra steps
  - No patching
- Must present a very low overhead
- Must support cross-compilers
- Must be smart enough to fall back on the local node if any risk
- Should ignore unreachable machines

=> ***distcc is exactly all this***

A few words on distcc

- Works either as a wrapper or in masqueraded mode :

```
$ ln -s /usr/bin/distcc x86_64-gcc47-linux-gnu-gcc
$ make -j 20 CC=$PWD/x86_64-gcc47-linux-gnu-gcc
```

- Automatically detects options involving the local node.
- Can use environment variables for nodes list
- Supports per-node usage limits
- Uses file-based node locking (no deamon)
- Detects dead nodes and can sometimes retry locally

Warnings on distcc

- Will not use remote nodes if gcov profiling is enabled

  => sed -i ‘s,.*\(CONFIG_GCOV_KERNEL\).*,\1=n,’ .config

- Pre-processing and linking are local and consume some time (approx 20-30%)

How to compare performance

- Depends on number of files, CPU count, parallelism, etc...
- Need to use a **portable** project as a reference to compare all machines, and stick to that version
- Count the project's line count to establish a metric in lines per second.
  - Hint: replace "gcc -c" with "gcc -S", build, and count non-empty lines not starting with "#".
  - Eg: 458870 lines for haproxy-6bcb0a84.

How to compare performance (...)

- Set cpufreq governor to “performance”
- Run multiple times (get at least 3 similar results)
- For more details on the method, please consult the wiki link at the end.

Suitable machines

The first question is :

What do we want to optimize ?

- Build performance for a given cost ?
- Number of nodes (*ie switch ports*) ?
- Hardware cost for a given performance level ?
- Hardware size / weight ?
- Power consumption / cooling / noise ?

Suitable machines (...)

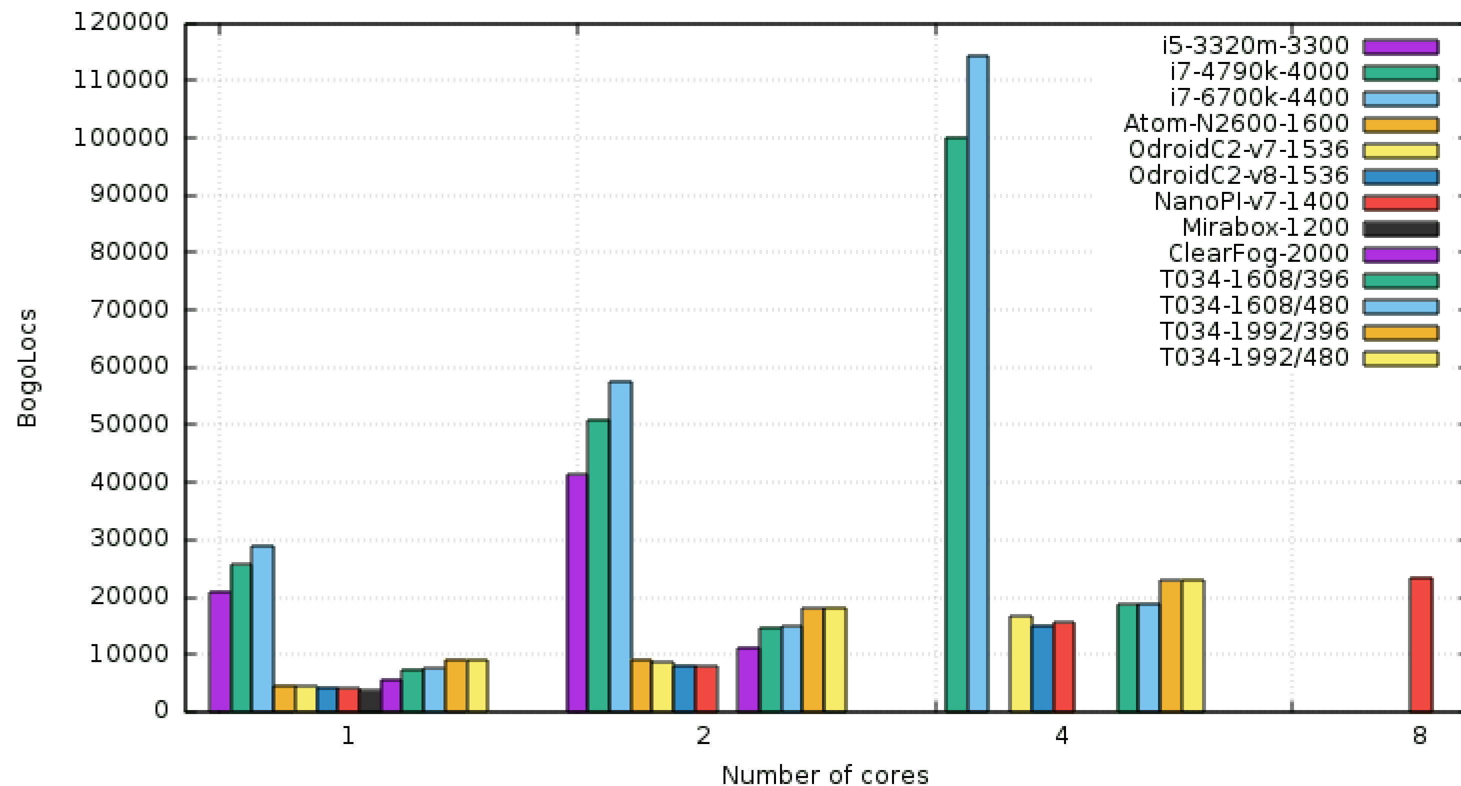
The second question is :

What impacts performance ?

- CPU architecture and family (AMD Phenom and later, intel core i3/i5/i7, ARM cortex A17 & A53 are efficient)
- DRAM latency (target high frequency and large busses)
- CPU caches sizes and latencies
- Storage access time if any (building in /dev/shm is better)
- Compiler build options (save ~10% by playing with ct-ng)

Suitable machines (...)

- The “BogoLocs” metric models expected performance numbers and avoids uselessly testing unsuitable hardware :



Suitable machines (...)

- BogoLocs only uses ramlat output
- It is surprizingly accurate for a prediction
- It reveals is that the most important speed limiting factors are the following, from most to least important :
  - DRAM latency (time to load a cache line)
  - L3 cache latency (if any)
  - L2 cache latency (if any)
  - CPU frequency
  - Core count.

Optimizing for performance

Always ensure the build nodes are properly used :

- Top/vmstat should almost always show ~100% CPU
- Local machine should almost never stay around 100% CPU
- Network must not be saturated

Real world example :

- Controller: Thinkpad T430s (core i5 2*3.1 GHz). Avg 70% CPU
- Build nodes: 4*CS-008 (RK3288 4*1.8 GHz). 95% CPU
- Network: 180 Mbps avg, peaks at 450 Mbps.
- Theorical limit for this controller: ~6 build nodes.

Optimizing for performance (...)

PC-type machines are the fastest and can be tuned :

- Fill all memory channels with fastest DRAM available (20-50% gains possible)
- Enable Hyperthreading when available (~50% gain max)
- Overclocking is a possibility (~15% gain max)
- Add more RAM to improve file caching
- Less gain over 8 cores due to memory bottlenecks

… but they (are expensive) and (are huge or noisy (or both)) and (draw a lot of power) but they are versatile.

***Hint**: chose motherboards, CPUs and RAM designed for gamers.*

Optimizing for the number of nodes

Use case : limited connectivity, limited build parallelism, boss willing to buy only one server, etc.

=> **Need for highest performance per node**

Dual-socket high-frequency, 8-core x86 machines with all RAM slots populated have a huge memory bandwidth and are not very expensive.

=> ***often bigger than the typical PC and draw more power.***

Optimizing for hardware costs

In 2016, a quad-core 1.2 GHz machine costs $8.
Eg: NanoPI NEO :

http://www.friendlyarm.com/index.php?route=product/product&path=69&product_id=132

Optimizing for hardware costs (...)

But does it really work ?

- Yes it does, but it will be around 16 times slower than a $800 PC.
- It may be even slower once it starts heating and throttles
- It is limited to 100 Mbps per node (ok for this one)

… and there are hidden costs

Hidden costs of very small hardware

- Shipping costs : ~$5 per machine
- Connectivity : a switch costs at least ~$2 per port
- Ethernet cable : ~$1 per cable
- A heatsink is needed to avoid throttling : $1
- A micro-SD card is needed to boot : $3
- A USB power supply is needed : $2

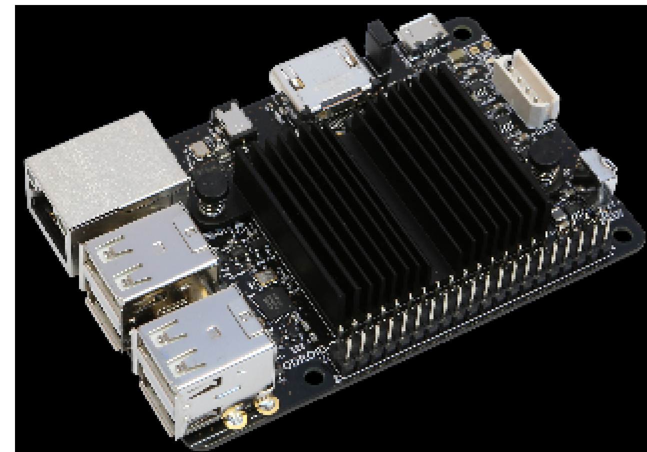
=> **Total:** ~$22 per machine for 1/16 of a $800 PC = $350 per PC equivalent, not very interesting

Pitfalls of very small hardware

- Large files will take 15-20 times as long as they take on a PC. A single large file can extend the build time.
- Quickly require some enclosures or rack mount options
- Heating, heating, heating...
- Stability issues : many vendors sell overclocked CPUs (eg: some OrangePI claim 1.6 GHz for this 1.2 GHz CPU)
- Sometimes advertised frequency cannot be reached

Mid-range hardware

- Often sold as “small PCs”
- $25-$60 price range
- CPUs up to quad-cortex A9 at 2 GHz
- Eg: NanoPI2-Fire and Odroid-C2:



Mid-range hardware (...)

Pros:

- Very interesting form factor, better cooling
- High performance density
- Some available in 8 A53 cores (eg: NanoPI-M3)

Cons:

- Comparatively more expensive per board
- Some platforms heat a lot (eg: Allwinner chips)
- Often forced to run their own kernel

High-range hardware

- Often sold as “set-top boxes” (STB)
- $50-$200 price range
- CPUs: 4-8 A17/A53 at 1.5-2.0 GHz
- Eg: MiQi, RKM-v5, CS-008 (clone) :

High-range hardware (...)

Pros:

- Very high performance density (up to ¼ of a $800 PC)
- Gigabit Ethernet connectivity (not always)
- On-board eMMC storage
- Often ultimately gets supported in mainline (eg: MiQi)

Cons:

- Often sold on Android, porting can be a pain (but fun)
- Setting up a chroot in android doesn’t always work due to selinux

High-range hardware (...)

Warnings:

- Varying build quality (eg: CS-008 clones with fake RAM chips)
- Varying software quality for cheaper clones
- Kernel lying on the real frequency
- Advertised power draw is lower than reality, need for a strong USB power supply.
- Thermal throttling may happen on heating, heatsinks are always needed even if sold without

Personal choice

My personal preference goes to these ones :

1. MiQi ($39, 4xA17 1.8 GHz, 1GB 64-bit DDR3-1600)

   => ~25k LoC/s

2. NanoPI-M3 ($35, 8xA53 1.4 GHz, 1GB 32-bit DDR3-1600)

   => ~21k LoC/s

3. Odroid-C2 ($35, 4xA53 1.5 GHz, 2GB 32-bit DDR3-912)

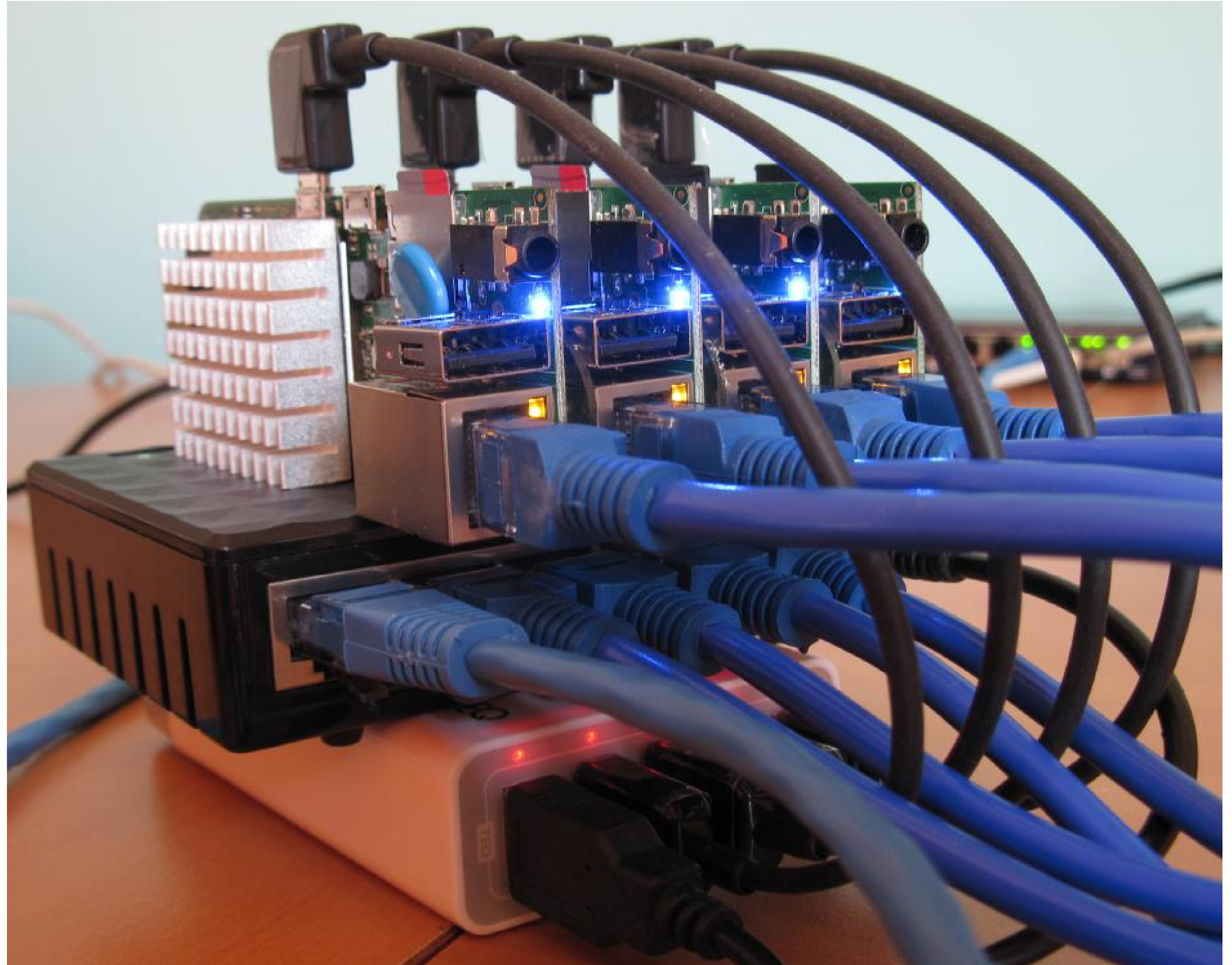
   => ~16k LoC/s

4. RKMv5/T034/CS008 ($60, 4xA17 1.6GHz, 2GB 32-bit DDR2-800)

   => ~18k LoC/s

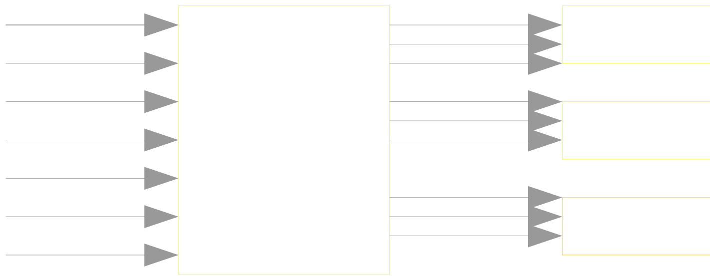
Current Implementation

4 CS-008 at 1.8 GHz, 5-port GigE switch, 5x2A PSU. Total cost: **$280** for 16 A17 cores and 8 GB RAM. Builds allmodconfig in **14 mn** vs **43 mn** on laptop.



Future research

- Experiment with distcc's “pump” mode
- Use haproxy in front of distccd :



- Avoid the switch for the portable version :
  - Try WiFi with LZO (unsuited without, 180-450 Mbps for 4 machines).
  - Try USB-NET on OTG ports with LZO over a USB hub (some USB power supplies include the hub).

More or less important hints

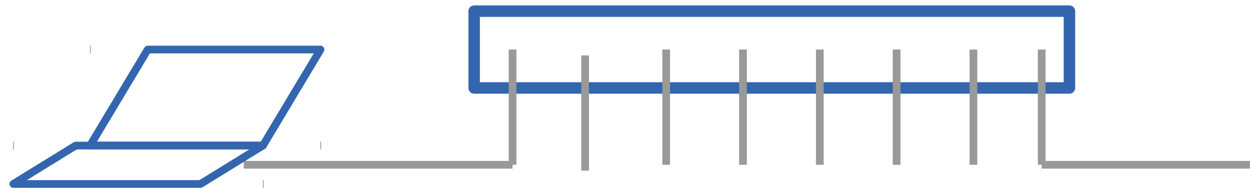
- Don’t focus on kernel optimization/support as long as it is able to quickly and reliably execute gcc.
- Any lightweight distro works (even a chroot inside Android if selinux is not too strict). Using “formilux” here (~10 MB).
- Check with “top” that there’s no CPU hog on the machine (lightdm, systemd)
- Use “ramlat” to measure the performance impact of graphics mode. Disable, reduce, rebuild without support, or kill Xorg.
- Unlocked core i7 are amazing when stuffed with fast DRAM
- Xeons and Atoms are not really worth the price.
- Cortex A17 is very strong (half a core i5 at 2/3 the frequency) but expensive
- Cortex A53 is 2/3 its performance and much cheaper but often provided with slow RAM and more cores
- Cooling: it’s up to you (passive, active, ...)

More or less important hints (...)

- configure leds as heartbeat to spot dead machines
- On a desk, daisy-chain clusters : 3 per 5-port switch, 6 per 8-port switch.
- Overclocking: sometimes OK, especially for build testing, or on “gamers” machines.
- Use power-meters to ensure the power usage remains below 10W per node for fanless operation

More or less important hints (...)

- configure leds as heartbeat to spot dead machines
- On a desk, daisy-chain clusters : 3 per 5-port switch, 6 per 8-port switch:



- Overclocking: sometimes OK, especially for build testing, or on “gamers” machines.
- Use power-meters to ensure the power usage remains below 10W per node for fanless operation

Useful links

- Distcc: https://github.com/distcc
- Crosstool-ng: http://crosstool-ng.org/
- Wiki page listing test reports of devices : http://wiki.ant-computing.com/Choosing_a_processor_for_a_build_farm
- mqmaker's MiQi board : https://forum.mqmaker.com/t/announcing-miqi-a-credit-card-sized-computer/371
- FriendlyARM's NanoPI boards : http://www.friendlyarm.com/
- Hardkernel's Odroid-C2 board : http://www.hardkernel.com/main/products/prdt_info.php
- Rikomagic's RKM-v5 device : http://www.rikomagic.co.uk/

That's all folks!

Thanks!

Questions / Comments / Jokes ?

Contact: Willy Tarreau <willy@haproxy.com>

HAProxy is hiring talented people. Contact us!