

External process synchronization for fun and profit

January 6, 2020

The LLVM project

- “a collection of modular and reusable compiler and toolchain technologies”
 - `llvm` – code generator and optimizer for many CPUs
 - `lldb` – debugger
 - `lld` – linker
 - `clang` – C/C++/Objective-C compiler
 - `clang-tidy` – C/C++/Objective-C static analysis tool
 - `clang-format` – C/C++/Objective-C code format tool
 - ...
- <https://llvm.org/>
- <https://github.com/llvm/llvm-project>

Let's build `llvm` and `clang`

```
$ git clone https://github.com/llvm/llvm-project.git
```

```
$ mkdir _build
```

```
$ cd _build
```

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

```
$ make
```


That didn't go so well...

- System resource usage is low
- Build takes cca. 5 hours – unacceptably long
- Let's try some tricks:
 - use `ccache`
 - use more `make` jobs
 - use `distcc` for distributed compilation

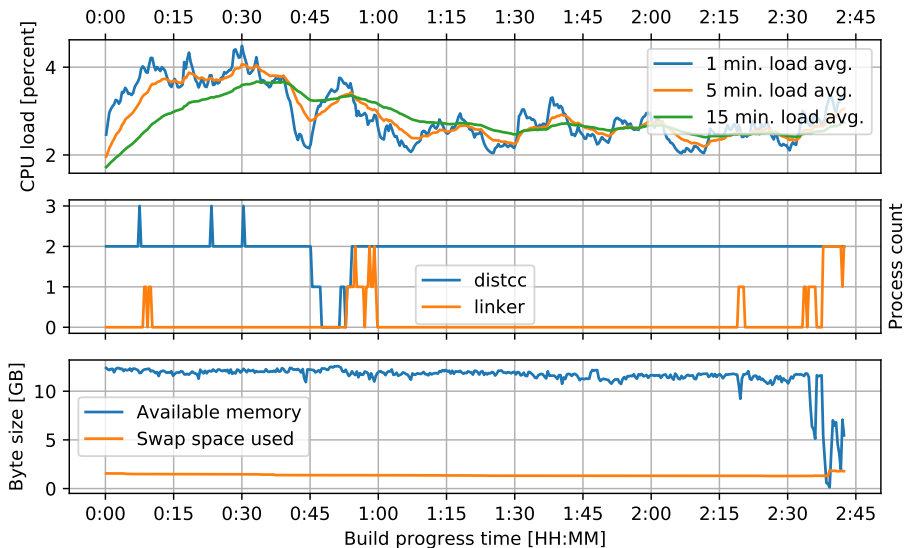
ccache – compiler cache

- “speeds up recompilation by caching previous compilations and detecting when the same compilation is being done again”
- <https://ccache.dev/>

distcc – distributed compilation for C and C++

- “distributes compilation of C or C++ code across several machines on a network”
- <https://github.com/distcc/distcc>

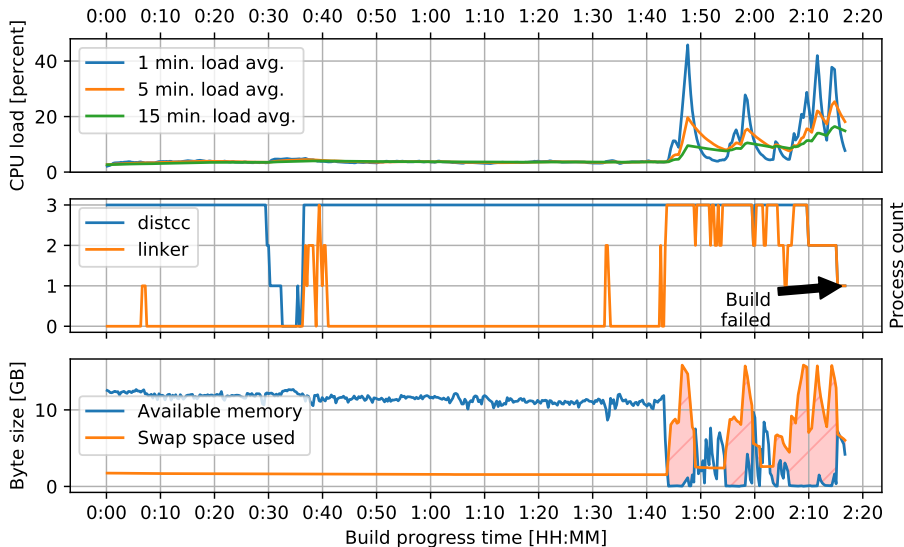
More jobs – `make clean && make -j 2`



That's somewhat better...

- System resource usage is still low
- Build takes more than 2 and half hours – still too long
- Parallelization shows potential

Keep going – `make clean && make -j 3`

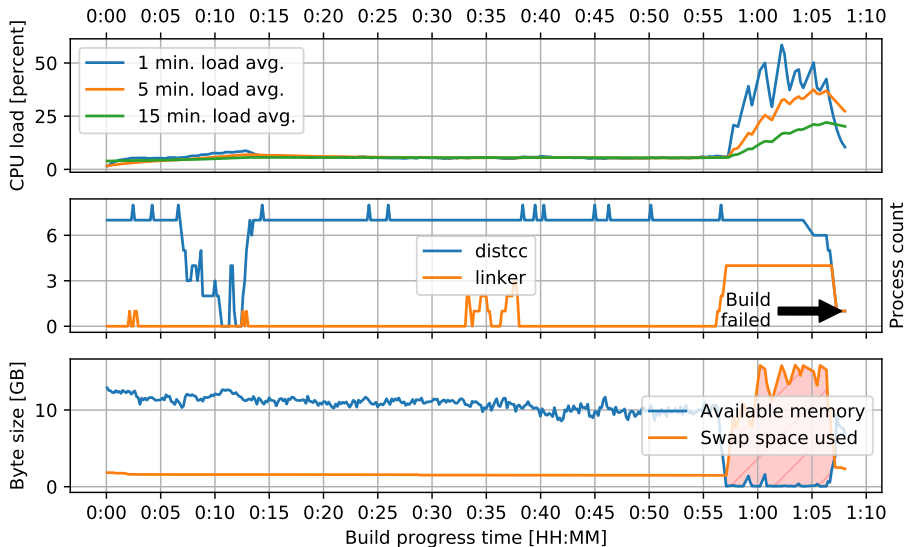


What happened?

- System usage starts to climb
 - especially memory usage
 - signs of correlation with linker execution
- Build *failed* after 2 hours 20 minutes – looooong
 - Linux OOM¹ process killer, kills some linker jobs
- Parallelization still shows potential

¹Out Of Memory

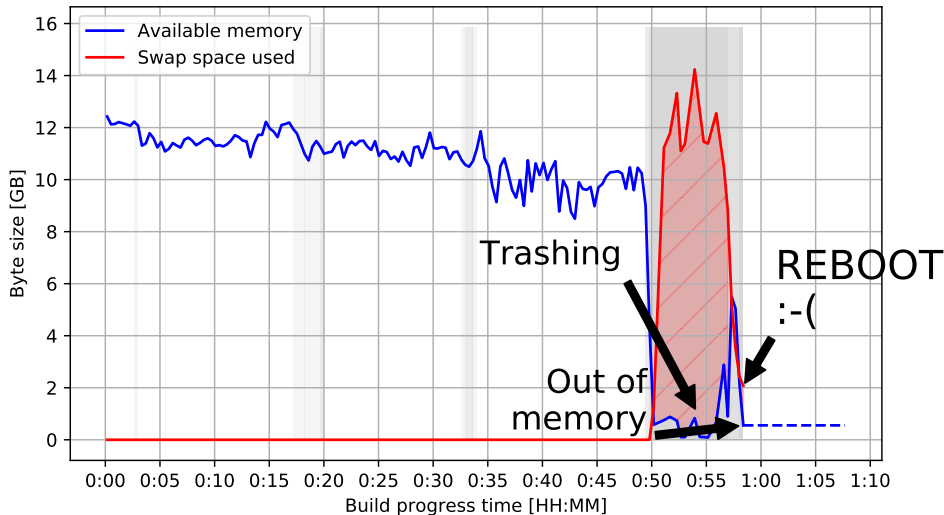
Meh. MOAR jobs! `make clean && make -j 7`



Not great, not terrible

- System usage very high toward the end of the build
 - especially memory usage
 - still correlated with linker execution
- Build *failed* after 1 hours 5 minutes – getting better
 - OOM process killer, kills some linker jobs
- Parallelization rules!

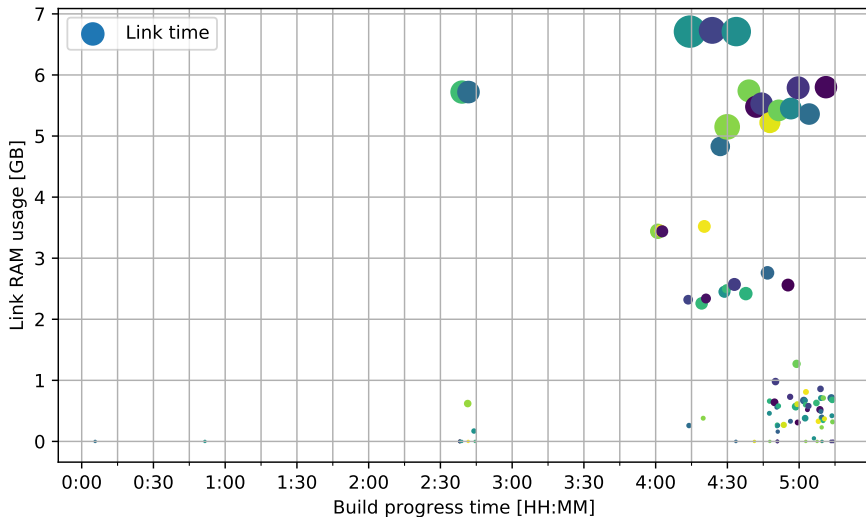
Still come cores left... `make clean && make -j 9`



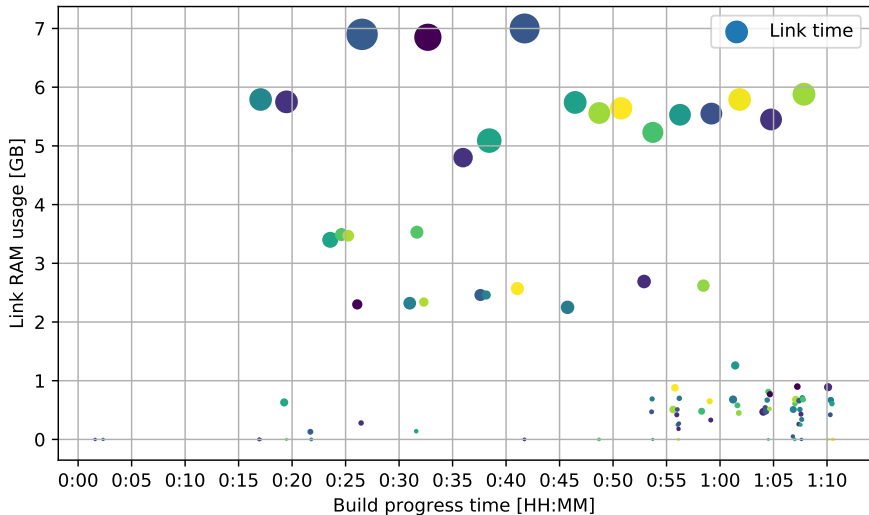
What have we learned so far

- Most of the build time is spent by compilation
- Compilation caching – good.
- Compilation parallelization – good.
 - We have not even really used `distcc` yet.
- Linking parallelization – bad! Why?
 - Let's have a closer look!

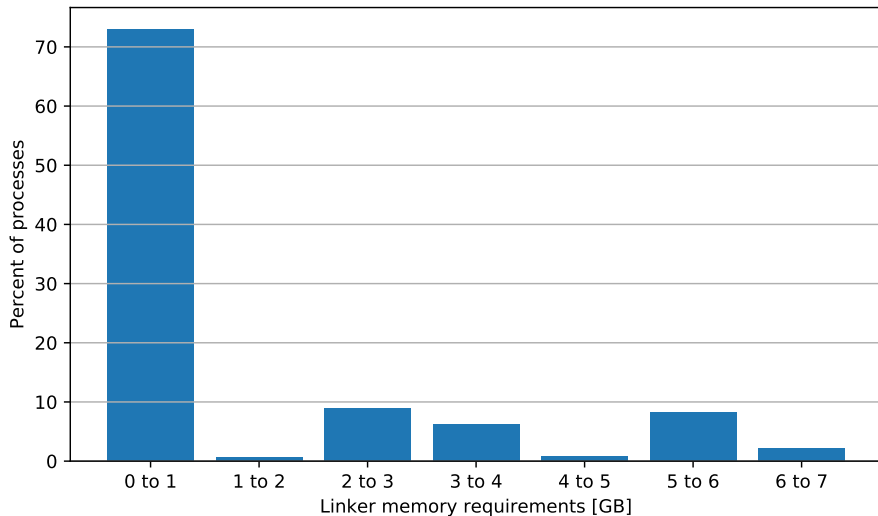
Linker memory usage vs link time – no cache



Linker memory usage vs link time – cached



Link target memory requirements – classification



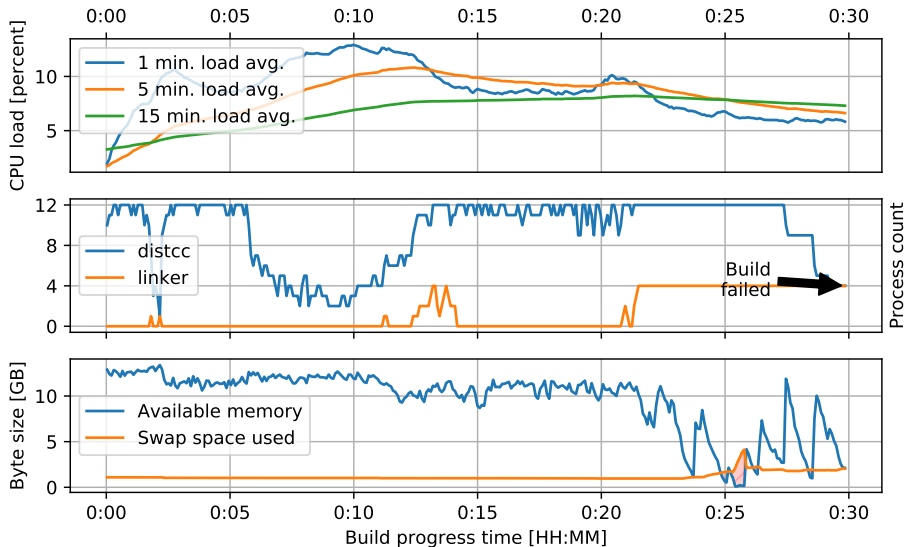
Linking `llvm` and `clang` in a nutshell

- Around 100 different link targets
 - Executables, Shared libraries
 - Many of them are small.
 - Few of them are **huge**.
- Most of the linking is done towards the end of the build process.
 - ⇒ Using many parallel jobs, multiple instances of linker run concurrently.
 - ⇒ Many big targets are linked at the same time.
 - ⇒ **OOM!**
- BTW: so much swapping is bad for the SSDs.

Making parallel linking work

- Let's try some additional tricks:
 - Use GNU `gold` instead of GNU `ld`.
 - Can link only in ELF format.
 - + Faster than `ld`.
 - + Uses less memory than `ld`.
 - Use `zram`.
 - a Linux kernel module for creating a compressed block device in RAM
 - can be used as swap device – part of swap is in compressed RAM
 - Make only some targets
 - Practical when developing and debugging.
 - Sometimes you just need to do `make all`.

```
zram + distcc + ccache + gold + make -j 12
```



If only...

- ... we could
 - prevent so many big targets from being linked at once,
 - prevent excessive swapping to disk,
- but still
 - use the hardware resources efficiently,
 - use available remote cores via `distcc` to full extent.

Let's synchronize the execution of `ld` / `gold`

- `atmost` to the rescue!
 - Simple utility written in C.
 - Takes a command-line (executable + arguments).
 - Waits until specified conditions are met.
 - Executes the command-line.
 - <https://github.com/matus-chochlik/various/atmost>

atmost – description

“The atmost utility allows to limit concurrent execution of a single executable or a set of executables, depending on various criteria like CPU load, memory and swap usage, thermal zone temperatures, current number of I/O operations, etc., or just by a maximum number.”

RTFM – man atmost

ATMOST(1)

General Commands Manual

ATMOST(1)

NAME

atmost - tool for limiting the concurrent execution of a specified executable.

SYNOPSIS

```
atmost [-v|--verbose] [-f|--file file-path]
        [-s|--socket socket-path] [-r|--reset]
        [-i|--sleep-interval seconds] [-c|--print-current]
        [-C|--print-all-current] [-n|--max-instances count]
        [-l|--max-cpu-load-1m percent]
        [-L|--max-cpu-load-5m percent] [-m|--min-avail-ram percent]
        [-M|--min-free-ram percent] [-S|--min-free-swap percent]
        [-p|--max-total-procs count] [-tc|--max-cpu-temp temp]
        [-tg|--max-gpu-temp temp] [-tb|--max-bat-temp temp]
        [-io|--max-io-ops count] [-nw|--max-nw-speed speed]
        [-snw|--slow-nw-speed speed] [-- executable [args...]]
```

```
atmost --help
```

`atmost -n NUMBER -- EXECUTABLE` – basic usage

- Limits the concurrent execution of `EXECUTABLE` to the specified `NUMBER` of instances.
- Uses an IPC semaphore set².
 - *Acquires*³ the semaphore before executing `EXECUTABLE`.
 - *Releases* the semaphore after executing `EXECUTABLE`.
 - `realpath`⁴ of `EXECUTABLE` is used as the semaphore set token⁵.

²see `man 2 semget`

³see `man 2 semop`

⁴see `man 3 realpath`

⁵see `man 3 ftok`

Using `atmost` – wrapper scripts

- In order to synchronize an `EXECUTABLE` with `atmost` :
 - Create a shell script with the name of `EXECUTABLE`.
 - From the script call `atmost` with appropriate arguments.
 - Put the wrapper script to `${PATH}`

```
$ vim /opt/bin/ld
```

```
#!/bin/bash  
atmost -n 2 -- /usr/bin/gold "${@}"
```

```
$ chmod u+x /opt/bin/ld
```

```
$ export PATH="/opt/bin:${PATH}"
```

```
atmost -n 2 -- ld
```

- + Empirically we have found that 2 instances of `ld` can run safely with 16GB RAM.
- + Simple
- + Low overhead
- Too coarse and restrictive
 - The majority of the `llvm` targets has low RAM requirements for linking.
 - Targets from other projects also have low linker RAM requirements.
 - We could safely use more parallel link jobs.

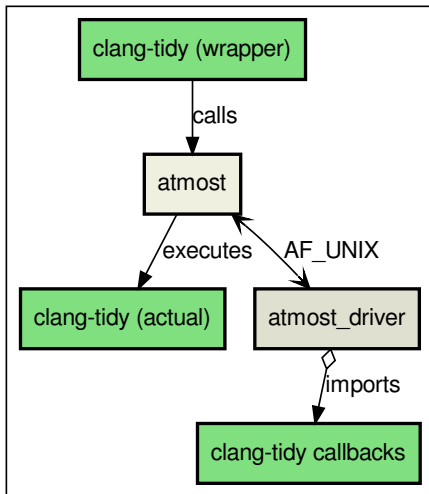
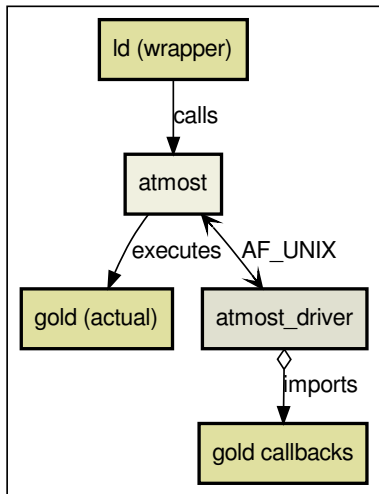
What if...

- ...we could **determine** if it is safe to start the linker per each invocation?
- Enter **atmost_driver**!
 - Running as a local **AF_UNIX** socket server.
 - **atmost** is the client (with the **--socket PATH** option).
 - When **atmost** is started, it connects to the *driver* and sends information about the wrapped **EXECUTABLE**.
 - When the *driver* determines that it's safe to run **EXECUTABLE**, it responds to **atmost**.
 - Then **atmost** executes **EXECUTABLE**.

Let's make it even more flexible

- `atmost_driver` callbacks:
 - The `atmost_driver` implements the reusable common server functionality.
 - The specific logic determining when it is safe to start executing, is implemented in a set of callbacks.
 - `load_callback_data`
 - `save_callback_data`
 - `process_initialized`
 - `let_process_go`
 - `process_finished`

atmost + atmost_driver + callbacks



Callback – `load_callback_data`

```
def load_callback_data():  
    resources = do_load_resources(...)  
    return resources
```

- Called once, when the `atmost_driver` is being started.
- Typically does startup initialization.
- Can load resources used by the other callbacks.

Callback – process_initialized

```
def process_initialized(resources, proc):
    # handle new process
    proc.set_callback_data(callback_data)
```

- Called when `atmost` sends the driver information about a new process.
- The `proc` parameter provides information about the new process.
 - PID, command-line, environment, working-directory, etc.
 - The callback can store its bookkeeping data in `proc`.

Callback – `let_process_go`

```
def let_process_go(resources, procs):  
    if can_process_start(resource, procs):  
        return True  
    return False
```

- Called repeatedly.
- Determines if a process can start executing the synchronized executable.
- The `procs` argument contains info about *all* currently managed processes, split into 3 groups.
 - `active` – processes that are already executing.
 - `waiting` – processes that are waiting for execution.
 - `current` – the process to be let go.

Callback – `process_finished`

```
def process_finished(resources, proc):  
    callback_data = proc.callback_data()  
    # cleanup info about the process
```

- Called when a process handled by the driver has finished.
- The callback can retrieve its bookkeeping data from `proc`.

Callback – `save_callback_data`

```
def save_callback_data(resources):  
    do_cleanup_resources(resources)
```

- Called once, when the `atmost_driver` is being shutdown.
- The `resource` parameter is the return value from `load_callback_data`.
- Can cleanup and/or save resources used and data generated during the run.

RTFM – `man atmost_driver`

ATMOST(1)

General Commands Manual

ATMOST(1)

NAME

`atmost_driver` - driver server for the `atmost` command.

SYNOPSIS

`atmost_driver [-s|--socket socket-path] [-c|--callbacks file-path]`
`[-u|--update interval]`

`atmost_driver -h`

`atmost_driver --help`

Back to linking...

- What if the memory usage can be determined from the linker command-line arguments?
 - 1) Run a lot of builds of various projects.
 - 2) Gather a lot of data
 - command line arguments,
 - input file sizes,
 - actual linker memory usage,
 - etc.
 - 3) ???
 - 4) Profit!

What exactly is step 3) ???

- Feed the gathered data into a machine learning model and train it.
- Use the trained ML model to **predict linker memory usage** from command-line arguments.
- Integrate the ML classifier into a `atmost_driver` callback script.
 - Keep track of available memory.
 - Predict the memory usage of new instances of linker.
 - Let the new linkers go only if there is enough available memory.

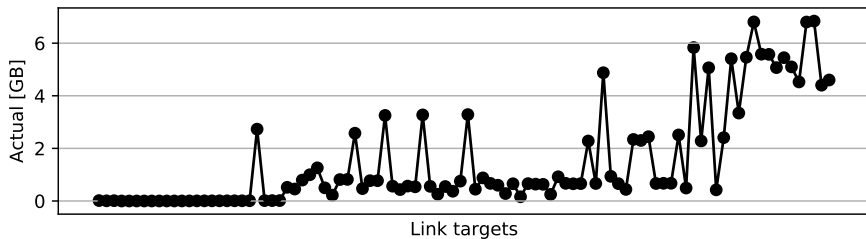
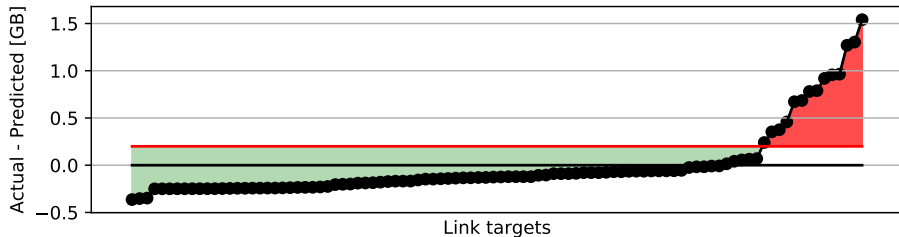
The ML part

- Uses the `neural_network.MLPClassifier` from the `sklearn`⁶ Python package.
- Inputs:
 - optimization level, the PIE⁷ flag,
 - count and combined size of input shared and static object files,
 - etc.
- Output:
 - the predicted memory requirements, as a multiple of 256MB chunks,
 - i.e. if the output is 5 the predicted size is $5 * 256[MB] = 1.25[GB]$.

⁶<https://scikit-learn.org/stable/documentation.html>

⁷position-independent executable

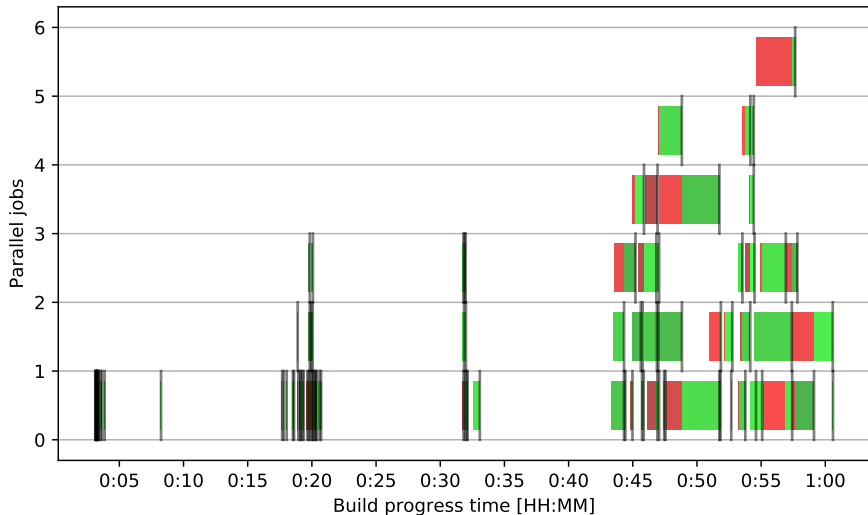
Classifier prediction accuracy



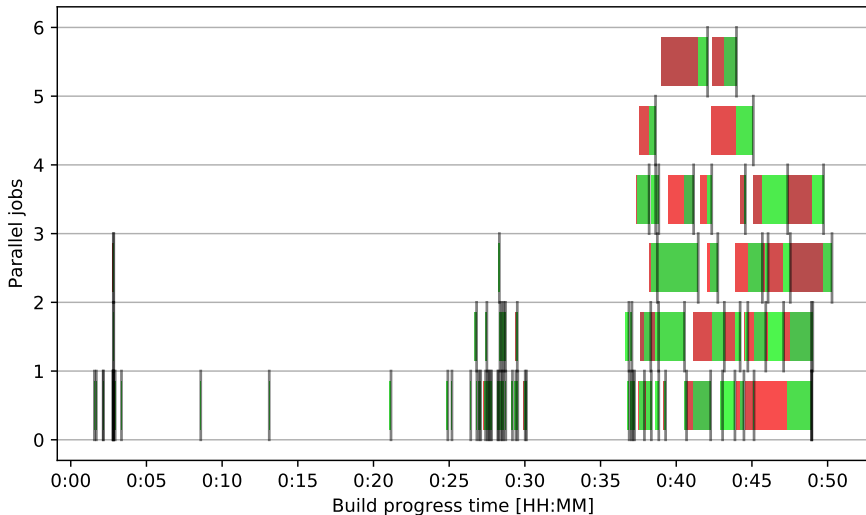
Integrating classifier with `atmost_driver`

- `load_callback_data` – load the trained ML model.
- `process_initialized` – use the model to predict memory usage.
- `let_process_go` – only if predicted linker memory usage is less than current available memory.
- `process_finished` – output actual memory usage that can be stored and used for additional model training.

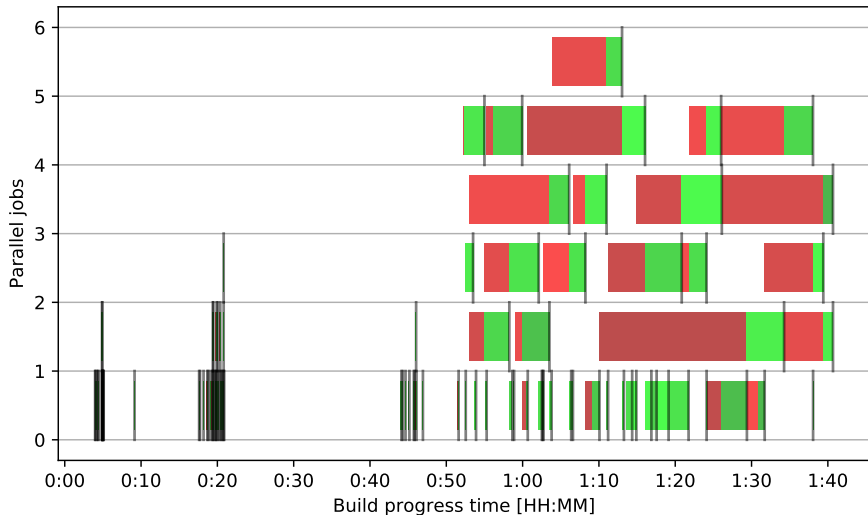
Linker schedule (no ccache, 16GB RAM) – `make -j 10`



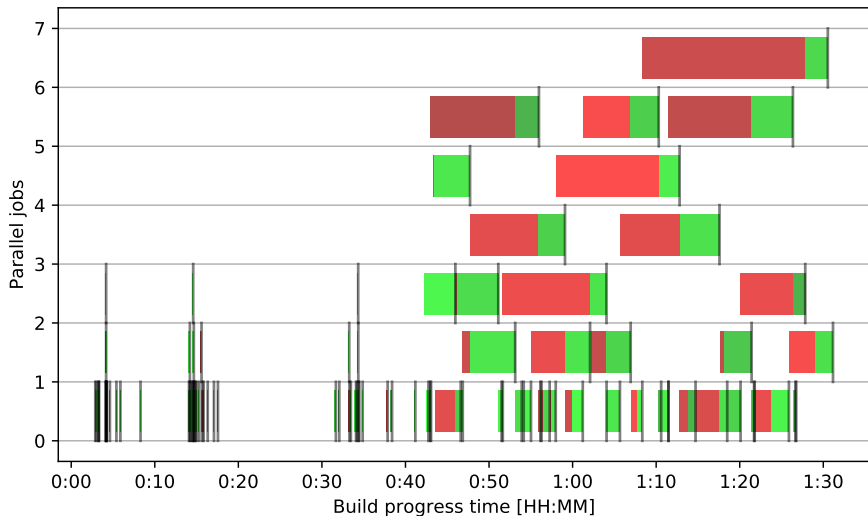
Linker schedule (no ccache, 16GB RAM) – `make -j 20`



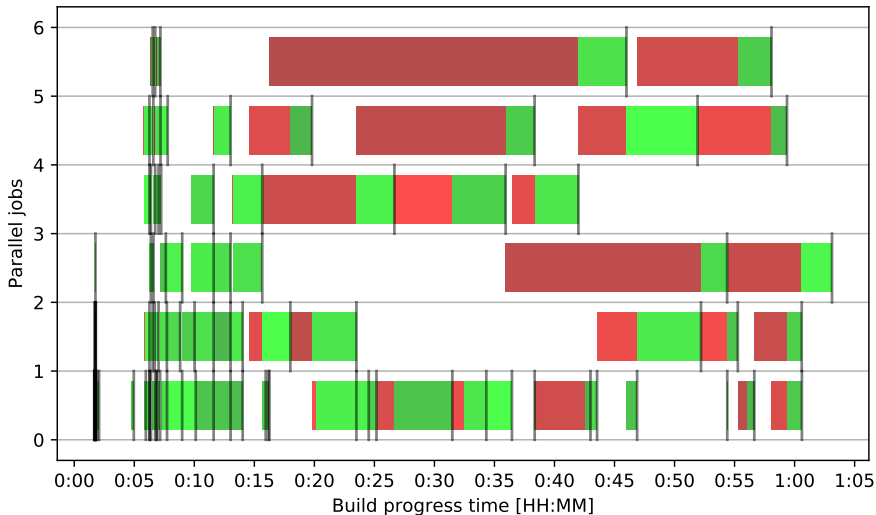
Linker schedule (no ccache, 8GB RAM) – `make -j 10`



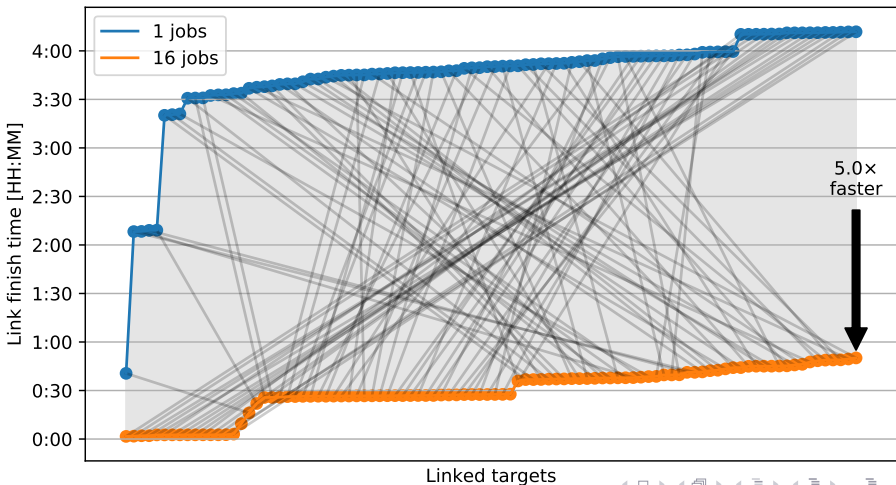
Linker schedule (no ccache, 8GB RAM) – `make -j 20`



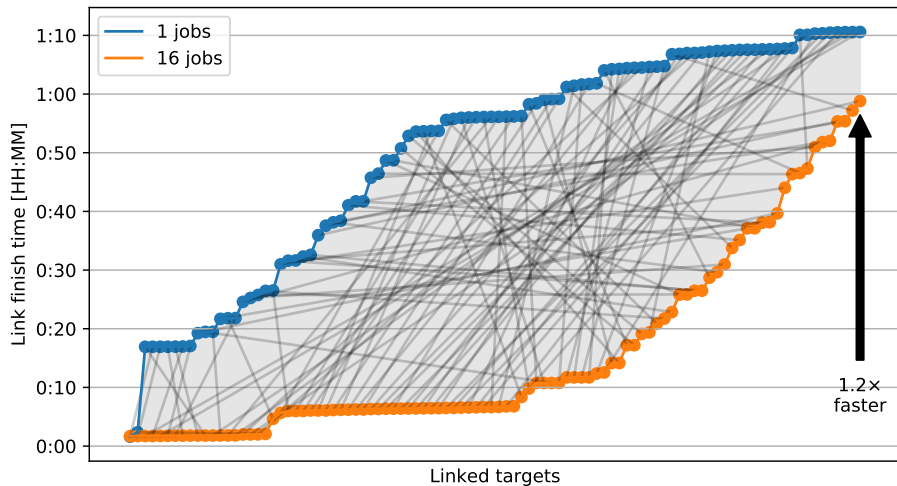
Linker schedule (ccached, 8GB RAM) – `make -j 20`



Link order vs. time (no ccache, 16GB, 4+12 cores, 16 jobs)



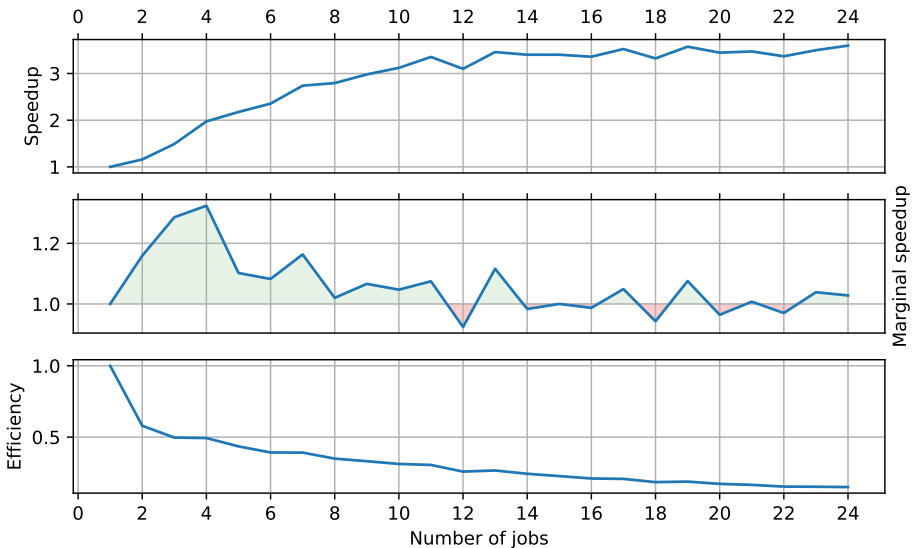
Link order vs. time (ccached, 8GB, 8+8 cores, 16 jobs)



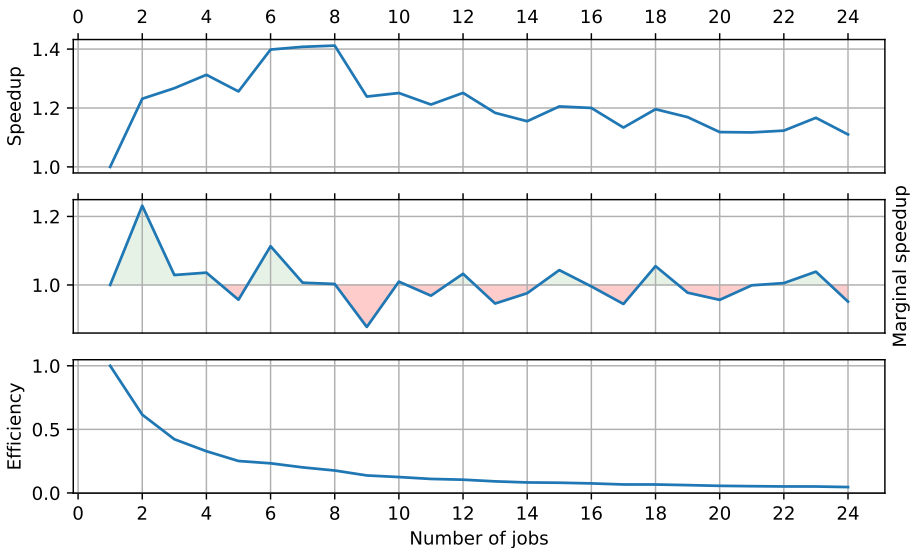
Parallelization statistic indicators

- *Run time with j jobs*
 - T_j
- *Speedup with j jobs*
 - $S_j = T_1 / T_j$
- *Marginal speedup with j jobs*
 - $M_j = T_{j-1} / T_j$
- *Efficiency with j jobs*
 - $E_j = S_j / j$

Parallelization statistics (no ccache, 8GB, 8+8 cores)



Parallelization statistics (ccached, 8GB, 8+8 cores)



Conclusions

- `atmost` is a highly flexible tool for synchronizing execution of parallel processes,
 - improves build times by allowing many parallel `make` jobs while serializing huge link jobs,
 - prevents system stalls, crashing and reboots due to low memory conditions,
 - makes building big projects safer,
 - allows to build `llvm` on systems with low RAM capacity,
 - reduces the amount of writes to swap disk partitions.

the *fun* part

- Learned new things:
 - new Python 3 features
 - Python *SciKit-Learn* package
 - Python *MatPlotLib* package
 - Secure `distcc` setup over `ssh`.
 - etc.

the *profit* part

- Stable and relatively quick `llvm` / `clang` builds.
- Reasonable HW usage.
- Saving the SSDs from excessive wear.
- Not getting random reboots during development and building.

the *moral* of the story

Go get more RAM!⁸

⁸if you can

Thank you!

Questions?

`https://github.com/matus-chochlik/
various/atmost/presentation/embeddmeet.pdf`

Extras

atmost – additional options

- Besides the basic semaphore-based synchronization and the customizable driver-based synchronization, **atmost** supports synchronization depending on:
 - 1 and 5 minutes average load,
 - amount of available RAM,
 - amount of free swap,
 - CPU temperature,
 - network speed,
 - battery status,
 - etc.

atmost – memory / swap usage

- `-m` or `--min-avail-ram`
- `-M` or `--min-free-ram`
- `-S` or `--min-free-swap`

Start executing only if at least 20% of RAM is available:

```
atmost -m 20 -- executable arguments...
```

Start executing only if at least 70% of swap is free:

```
atmost -S 70 -- executable arguments...
```

atmost – total process count

- `-p` or `--max-total-procs`

Start executing only if the number of currently running processes is ≤ 1000 :

```
atmost -p 1000 -- executable arguments...
```

atmost – thermal zone temperatures

- `-tc` or `--max-cpu-temp`
- `-tg` or `--max-gpu-temp`
- `-tb` or `--max-bat-temp`

Start executing only if the CPU temperature is less than or equal to 70°C:

```
atmost -tc 70 -- executable arguments...
```

Start executing only if the battery temperature is less than or equal to 55.5°C:

```
atmost -tb 55.5 -- executable arguments\  
...
```

atmost – modifiers

- Modifiers allow to change the limits by a specified amount under special conditions.
 - when running on battery,
 - when connected to “slow” network,
 - when disconnected from network.

atmost – on A/C vs. on battery

- `-batt` or `--on-battery`

When on A/C, start only if 1 minute average load is less than or equal to 15%,
when on battery, start only if average load is less than or equal to 5% = (15 - 10):

```
atmost -l 15 -batt 10 -- executable \
arguments...
```

atmost – combining limits and modifiers

When on A/C, start only if 1 minute load is $\leq 20\%$ and 5 minutes load is $\leq 10\%$,
when on battery, start only if 1 minute load is $\leq 8\%$ and 5 minutes load is $\leq 4\%$:

```
atmost -l 20 -batt 12 -L 10 -batt 6 -- \
executable ...
```

Limiting concurrent instances of clang-tidy

- Is it useful to limit concurrent execution of clang-tidy ?
 - 8 core CPU,
 - 2 local cores for compiling,
 - 4 local cores for compiling,
 - 6 local cores for compiling,
 - the rest used by clang-tidy .
- Measured on the test-suite of the OGLplus⁹ project.

⁹<http://oglplus.org/>

```
atmost -n  $\{N\}$  clang-tidy " $\{@\}$ "
```

