The definitive guide to make software fail on ARM64

Ignat Korchagin

@secumod
$ whoami

- Performance and security at Cloudflare
- Passionate about security and crypto
- Enjoy low level programming
Initial ARM64 integration
Initial integration in the DC
Consider your developers
Building packages for ARM64

production arch != developer arch
Building packages for ARM64
Compiler

mysrc.c
Compiler

- mysrc.c
- x86
Compiler

```
mysrc.c
```

```
x86
```

```
gcc
```
Compiler

mysrc.c → gcc → x86 → x86 → x86 → x86
Compiler

mysrc.c → gcc → x86 → mybin

mybin

mybin

mybin
Cross-compiler

- `mysrc.c`
- `x86`
Cross-compiler

mysrc.c → cross-gcc → x86
Cross-compiler

```plaintext
Cross-compiler

```

```plaintext
mysrc.c -> cross-gcc
```

```
arm64
```

```
arm64
```

```
arm64
```

```
arm64
```

```
arm64
```
Cross-compiler

```
mysrc.c  →  cross-gcc
        /   \
      /     \
    /       \
arm64  →  arm64  →  arm64
  mybin      mybin      mybin
```
Cross-compiler

- mysrc.c
- cross-gcc
- arm64
- mybin
- arm64
- mybin
- arm64
- mybin
Cross-compiler terminology

- host - architecture, where the compiler runs
Cross-compiler terminology

- **host** - architecture, where the compiler runs
- **target** - architecture, for which the compiler generates machine code
Cross-compiler terminology

- **host** - architecture, where the compiler runs
- **target** - architecture, for which the compiler generates machine code
- when host == target, it is “native” compilation
  - subset of a more general cross-compilation
cross-compiling example

ignat@dev:~$ gcc -static -o mybin mysrc.c
cross-compiling example

ignat@dev:~$ gcc -static -o mybin mysrc.c

ignat@dev:~$ readelf -h mybin | grep -i machine

  Machine: Advanced Micro Devices X86-64
cross-compiling example

ignat@dev:~$ gcc -static -o mybin mysrc.c
ignat@dev:~$ readelf -h mybin | grep -i machine
  Machine: Advanced Micro Devices X86-64
ignat@dev:~$ ./mybin
Hello, world!
cross-compiling example

ignat@dev:~$ gcc -static -o mybin mysrc.c
ignat@dev:~$ readelf -h mybin | grep -i machine
    Machine:                                         Advanced Micro Devices X86-64
ignat@dev:~$ ./mybin
Hello, world!
ignat@dev:~$ aarch64-linux-gnu-gcc -static -o mybin mysrc.c
cross-compiling example

ignat@dev:~$ gcc -static -o mybin mysrc.c
ignat@dev:~$ readelf -h mybin | grep -i machine
   Machine:                           Advanced Micro Devices X86-64
ignat@dev:~$ ./mybin
Hello, world!

ignat@dev:~$ aarch64-linux-gnu-gcc -static -o mybin mysrc.c
ignat@dev:~$ readelf -h mybin | grep -i machine
   Machine:                           AArch64
cross-compiling example

ignat@dev:~$ gcc -static -o mybin mysrc.c
ignat@dev:~$ readelf -h mybin | grep -i machine
  Machine:                           Advanced Micro Devices X86-64
ignat@dev:~$ ./mybin
Hello, world!

ignat@dev:~$ aarch64-linux-gnu-gcc -static -o mybin mysrc.c
ignat@dev:~$ readelf -h mybin | grep -i machine
  Machine:                           AArch64
ignat@dev:~$ ./mybin
bash: ./mybin: cannot execute binary file: Exec format error
Compile time problems
Common misconception

ASSEMBLY

ASSEMBLY EVERYWHERE
CC: hardcoded architecture-specific flags

Symptom:

- broken both native and cross builds
CC: hardcoded architecture-specific flags

Symptom:

- broken both native and cross builds
- `gcc: error: unrecognized command line option `-msse2'`
CC: hardcoded architecture-specific flags

Symptom:

● broken both native and cross builds
● \texttt{gcc: error: unrecognized command line option \textsc{--msse2}}

Cause:

● hardcoded architecture-specific flags in the build system
CC: hardcoded architecture-specific flags

Symptom:
- broken both native and cross builds
- \texttt{gcc: error: unrecognized command line option ‘-msse2’}

Cause:
- hardcoded architecture-specific flags in the build system
- \texttt{CFLAGS := ... -msse2 ...} or \texttt{CFLAGS += -msee2 ...}
CC: hardcoded architecture-specific flags

Developers:

- put architecture-specific flags in a separate variable, one for each architecture
CC: hardcoded architecture-specific flags

Developers:

- put architecture-specific flags in a separate variable, one for each architecture

# Makefile

TARGET_ARCH := ... # somehow identify the target architecture
CFLAGS_x86_64 := -msse2 ...
CFLAGS_arm64 := -mabi=lp64 ...
TARGET_CFLAGS += CFLAGS_$\(TARGET_ARCH\)
CC: no separation between host and target flags

Symptom:
- broken cross build
CC: no separation between host and target flags

Symptom:

- broken cross build
- usually happens, when the compiler output needs additional post-processing (ex. format conversion)
CC: no separation between host and target flags

Symptom:
- broken cross build
- usually happens, when the compiler output needs additional post-processing (ex. format conversion)
- post-processing tool source is part of the project
CC: no separation between host and target flags

Symptom:

- broken cross build
- usually happens, when the compiler output needs additional post-processing (ex. format conversion)
- post-processing tool source is part of the project
- `gcc: error: unrecognized command line option ‘-msse2’`
CC: no separation between host and target flags
CC: no separation between host and target flags

sources → $(TARGET_CFLAGS) → target gcc
CC: no separation between host and target flags

sources \(\rightarrow\) target gcc \(\rightarrow\) intermediate artifacts

$\text{TARGET_CFLAGS}$
CC: no separation between host and target flags

sources -> $(TARGET_CFLAGS) -> target gcc -> intermediate artifacts

tool sources
CC: no separation between host and target flags

sources

$({TARGET_CFLAGS})

1

target gcc

1

intermediate artifacts

2

$(HOST_CFLAGS)

host gcc

tool sources
CC: no separation between host and target flags

1. sources -> target gcc
   - $(TARGET_CFLAGS)
2. tool sources -> host gcc
   - $(HOST_CFLAGS)

1. target gcc -> intermediate artifacts
2. host gcc -> additional tools
CC: no separation between host and target flags

1. $(TARGET_CFLAGS)
2. $(HOST_CFLAGS)
3. Additional tools
CC: no separation between host and target flags

sources

$\text{TARGET\_CFLAGS}$

1

target gcc

$\text{HOST\_CFLAGS}$

2

additional tools

3

final artifacts

2

host gcc

3

intermediate artifacts
CC: no separation between host and target flags

Cause:

- $(CFLAGS) use instead of $(TARGET_CFLAGS) and $(HOST_CFLAGS)
CC: no separation between host and target flags

Cause:

● $(CFLAGS) use instead of $(TARGET_CFLAGS) and $(HOST_CFLAGS)

● use of some $(ADDITIONAL_CFLAGS) which are based either only on the target or the host
  ○ see the usage of $(WORKAROUND_CFLAGS) in the iPXE build system: https://github.com/ipxe/ipxe
CC: no separation between host and target flags

WORKAROUND_CFLAGS := ... # based on target
... 
TARGET_CFLAGS += $(WORKAROUND_CFLAGS)
... 
HOST_CFLAGS += $(WORKAROUND_CFLAGS)
CC: no separation between host and target flags

Developers:

- put architecture-specific flags in a separate variable, one for each architecture
CC: no separation between host and target flags

Developers:

- put architecture-specific flags in a separate variable, one for each architecture
- always prefix any compiler/linker options with `TARGET_` or `HOST_`
  - `$(WORKAROUND_CFLAGS)`, `$(TARGET_WORKAROUND_CFLAGS)` and `$(HOST_WORKAROUND_CFLAGS)`
  - use `$(COMMON_CFLAGS)` if needed
CC: no separation between host and target flags

DevOps:

● provide the tools/support to test cross-compilation in the CI
  ○ x86 to arm64 is generally a good start
CC: no separation between host and target flags

DevOps:

● provide the tools/support to test cross-compile in the CI
  ○ x86 to arm64 is generally a good start
● lint project build systems for non-prefixed variable definitions
Slower build times - it’s a feature!
Symptom:
● broken artifacts
CC: reuse of host binaries in target artifacts

Symptom:

- broken artifacts
- usually happens, when the compiler output needs additional post-processing (ex. format conversion)
Symptom:

● broken artifacts
● usually happens, when the compiler output needs additional post-processing (ex. format conversion)
● post-processing tool source is part of the project
CC: reuse of host binaries in target artifacts

Symptom:

- broken artifacts
- usually happens, when the compiler output needs additional post-processing (ex. format conversion)
- post-processing tool source is part of the project
- post-processing tool is also released as an artifact
CC: reuse of host binaries in target artifacts

Symptom:
● broken artifacts
● usually happens, when the compiler output needs additional post-processing (ex. format conversion)
● post-processing tool source is part of the project
● post-processing tool is also released as an artifact
● ./fixdep: cannot execute binary file: Exec format error
CC: reuse of host binaries in target artifacts

Cause:

- incorrect usage of $(HOST_CC) vs $(TARGET_CC)
CC: reuse of host binaries in target artifacts

Cause:

- Incorrect usage of $(HOST_CC) vs $(TARGET_CC)
- Incorrect build dependency declaration
  - “make” may consider the dependency, built with $(HOST_CC) already satisfied, when doing the target build and not rebuild it with $(TARGET_CC)
CC: reuse of host binaries in target artifacts

Cause:

- incorrect usage of \$(HOST\_CC) vs \$(TARGET\_CC)
- incorrect build dependency declaration
  - "make" may consider the dependency, built with \$(HOST\_CC) already satisfied, when doing the target build and not rebuild it with \$(TARGET\_CC)
- example: vanilla Linux kernel Debian packaging
  - broken "linux-headers" .deb package when cross-compiling
CC: reuse of host binaries in target artifacts

sources → target gcc

sources → host gcc

intermediate artifacts → additional tools → final artifacts

$\text{TARGET_CFLAGS}$

$\text{HOST_CFLAGS}$
CC: reuse of host binaries in target artifacts

sources → target gcc → intermediate artifacts

$\text{TARGET_CFLAGS}$

$\text{HOST_CFLAGS}$

tool sources → host gcc → additional tools → final artifacts → extra packaging

1. $(\text{TARGET_CFLAGS})$
2. $(\text{HOST_CFLAGS})$
3. 
4. 

extra packaging
CC: reuse of host binaries in target artifacts

- example: tools to build modules in Linux
- used on host, when doing the main build
- packaged into "linux-headers" .deb package
- `./fixdep: cannot execute binary file: Exec format error`

```
$(TARGET_CFLAGS) sources -> target gcc -> intermediate artifacts

$(HOST_CFLAGS) tool sources -> host gcc -> additional tools -> final artifacts
```

extra packaging
CC: reuse of host binaries in target artifacts

Developers:

- ensure all target artifacts are processed with 
  \$ (TARGET_CC)
Developers:

- ensure all target artifacts are processed with $(TARGET_CC)$
- put host and target output in different directories
  - clearly shows which artifacts are not compiled either for host or target architecture
  - ensures “make” does not consider target dependency satisfied, if only the host version was built, because of different filesystem paths

CC: reuse of host binaries in target artifacts
CC: reuse of host binaries in target artifacts

DevOps:

- provide the tools/support to test cross-compilation in the CI
  - x86 to arm64 is generally a good start
DevOps:

- provide the tools/support to test cross-compilation in the CI
  - x86 to arm64 is generally a good start
- inspect the final artifacts for anomalies
  - for example, there should be no x86 executables in the arm64 .deb package
Runtime problems
Out of memory with plenty of memory

Symptom:

- the process complains about not being able to allocate memory
Out of memory with plenty of memory

Symptom:

● the process complains about not being able to allocate memory
● there is plenty of free memory in the system
Out of memory with plenty of memory

Symptom:

- the process complains about not being able to allocate memory
- there is plenty of free memory in the system
- the process is using mmap syscall for file I/O
  - most database workloads
Out of memory with plenty of memory

Symptom:

- the process complains about not being able to allocate memory
- there is plenty of free memory in the system
- the process is using mmap syscall for file I/O
  - most database workloads
- **ENOMEM: Cannot allocate memory**
32-bit vs 64-bit

- 32-bit allows to address only up to 4GB
32-bit vs 64-bit

- 32-bit allows to address only up to 4GB

- 64-bit allows to address up to 17179869184GB
  - or “more than enough...”
The cake is a lie
Linux process virtual memory map (x86)

- **user-space**
  - (0x0000000000000000-0x00007fffffff)

- **huge hole**

- **kernel**
  - (0xffffffff800000000000-0xffffffffffffffff)

https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
Linux process virtual memory map

- actually you can have only 47-bit addresses in user-space on x86_64
  - so it is only 131072GB compared to promised 17179869184GB
Linux process virtual memory map

- actually you can have only 47-bit addresses in user-space on x86_64
  - so it is only 131072GB compared to promised 17179869184GB
- on arm64 you get only 39-bit addresses if you take Linux defaults
  - only 512GB addressable space

https://www.kernel.org/doc/Documentation/arm64/memory.txt
Linux process virtual memory map

Developers:

- try to avoid using unbounded memory mappings
Linux process virtual memory map

Developers:

- try to avoid using unbounded memory mappings
- try to identify the upper bound of the user-space addressable space and compare to the mapped file size
Linux process virtual memory map

Developers:

- try to avoid using unbounded memory mappings
- try to identify the upper bound of the user-space addressable space and compare to the mapped file size

DevOps:

- make sure to review your second architecture kernel memory layout config
  - you might need to recompile the kernel
Linux process virtual memory map (cont.)

- recompiled the arm64 kernel with 48-bit user-space addresses (256TB space)
Linux process virtual memory map (cont.)

- recompiled the arm64 kernel with 48-bit user-space addresses (256TB space)
- some workloads started to crash randomly
Linux process virtual memory map (cont.)

- recompiled the arm64 kernel with 48-bit user-space addresses (256TB space)
- some workloads started to crash randomly
- traced down to Lua code
LuaJIT lightuserdata

• simple “efficient” C-interface
LuaJIT lightuserdata

- simple “efficient” C-interface
- operates directly on C-pointers
LuaJIT lightuserdata

- simple “efficient” C-interface
- operates directly on C-pointers
- uses (supposedly unused) upper bits of the address to store some metadata
  - 0x00007fffffff

https://github.com/LuaJIT/LuaJIT/blob/f5d424afe8b9395f0df05aba905e0e1f6a2262b8/src/lj_obj.h#L173-L193
LuaJIT lightuserdata assumptions

/* Internal object tags.
 **
 ** Internal tags overlap the MSW of a number object (must be a double).
 ** Interpreted as a double these are special NaNs. The FPU only generates
 ** one type of NaN (0xffff_0000_0000_0000). So MSWs > 0xffff0000 are available
 ** for use as internal tags. Small negative numbers are used to shorten the
 ** encoding of type comparisons (reg/mem against sign-ext. 8 bit immediate).
 **
 **
 ** ---MSW---.---LSW---
 **
 ** primitive types | itype |   |
 ** lightuserdata   | itype | void * | (32 bit platforms)
 ** lightuserdata   | ffff | void *  | (64 bit platforms, 47 bit pointers)
 ** GC objects      | itype | GCreff  |
 ** int (LJ_DUALNUM)| itype | int    |
 ** number          | ---------double---------
 **
 ** ORDER LJ_T
 ** Primitive types nil/false/true must be first, lightuserdata next.
 ** GC objects are at the end, table/userdata must be lowest.
 ** Also check lj_ir.h for similar ordering constraints.
 */

/*
 */
Linux process virtual memory

Developers:

- state assumptions in code, not comments
  - check assumptions early and error out with a meaningful error message
Linux process virtual memory

Developers:

- state assumptions in code, not comments
  - check assumptions early and error out with a meaningful error message
- don’t over optimise
  - provide a fallback (less optimal) generic implementation
Developers:

● state assumptions in code, not comments
  ○ check assumptions early and error out with a meaningful error message
● don’t over optimise
  ○ provide a fallback (less optimal) generic implementation

DevOps:

● ditto
Pagesize

WHEN YOU NEED TO OPTIMISE SOMETHING

PAGESIZE
Pagesize

- a minimum discrete block of volatile memory
Pagesize

- a minimum discrete block of volatile memory
- many database-like workloads try to keep track of allocated pages
  - faster memory access
  - avoid memory fragmentation
  - efficient memory reuse
Pagesize

Symptom:

- the process uses much more memory on secondary architecture
Pagesize

Symptom:

● the process uses much more memory on secondary architecture
● otherwise, working as intended
  ○ although it depends how aggressive the code is with memory management
Pagesize

Cause:

- the process has hardcoded page size in code
Pagesize

Cause:

- the process has hardcoded page size in code
- the target architecture has a different page size
  - arm64 may have 4k, 16k or 64k pages

https://www.kernel.org/doc/Documentation/arm64/memory.txt
Pagesize 4k

managed pages

page 1 ── page 2 ── page 3
Pagesize

Developers:

- `#define PAGE_SIZE 4096`
  - ~14k+ exact matches on GitHub
Pagesize

Developers:

- `#define PAGE_SIZE 4096`
  - ~14k+ exact matches on GitHub
- `long page_size = sysconf(_SC_PAGESIZE);`
Pagesize

Developers:

- #define PAGE_SIZE 4096
  - ~14k+ exact matches on GitHub
- long page_size = sysconf(_SC_PAGESIZE);

DevOps:

- monitor process memory usage on different architectures
Filesystem block size

- like pagesize, but for files
Filesysterm block size

- like pagesize, but for files
- minimum amount any piece of data can occupy on disk, so determines physical file size
  - even 1 byte file will occupy at least “block” bytes
Filesystem block size

● like pagesize, but for files
● minimum amount any piece of data can occupy on disk, so determines physical file size
  ○ even 1 byte file will occupy at least “block” bytes
● multiple of the underlying block device block size
  ○ typical values are 512 bytes or 4k
Filesystem block size

- like pagesize, but for files
- minimum amount any piece of data can occupy on disk, so determines physical file size
  - even 1 byte file will occupy at least “block” bytes
- multiple of the underlying block device block size
  - typical values are 512 bytes or 4k
- mostly useful for sparse files

Sparse files

- Areas with Real Data which occupy physical disk space
- Holes - Sparse Zeros which don't occupy physical disk space

Logical File Size

Physical File Size
Filesystem block size

Symptom:

● the sparse file test fails on arm64
  ○ https://github.com/capnproto/capnproto
Filesystem block size

Symptom:

- the sparse file test fails on arm64
  - [https://github.com/capnproto/capnproto](https://github.com/capnproto/capnproto)
- the test fails only, when the test suite is run from tmpfs
Filesystem block size

Cause:

- the process has hardcoded block size in code
Filesystem block size

Cause:

● the process has hardcoded block size in code
● on memory-backed filesystems block size == page size
  ○ arm64 may have 4k, 16k or 64k pages

https://www.kernel.org/doc/Documentation/arm64/memory.txt
Filesystem block size

Developers:

```
#define BLOCK_SIZE 4096
```
Filesystem block size

Developers:

- `#define BLOCK_SIZE 4096`
- `stat("/the/file", &stats); blksize_t
  block_size = stats.st_blksize;`
Conclusions

- Even “portable” code with no assembly can fail in many ways on a different architecture.

- For developers:
  - Don’t over optimise, provide fallback implementations.
  - Don’t rely on assumptions and test them in code if you have to.
  - Provide meaningful error messages.

- For devops:
  - Ensure the CI environment can test diverse architectures and configurations.
  - Provide tools/linters to enforce best-practices in code and build scripts.
ARM64 in production
Thank you!