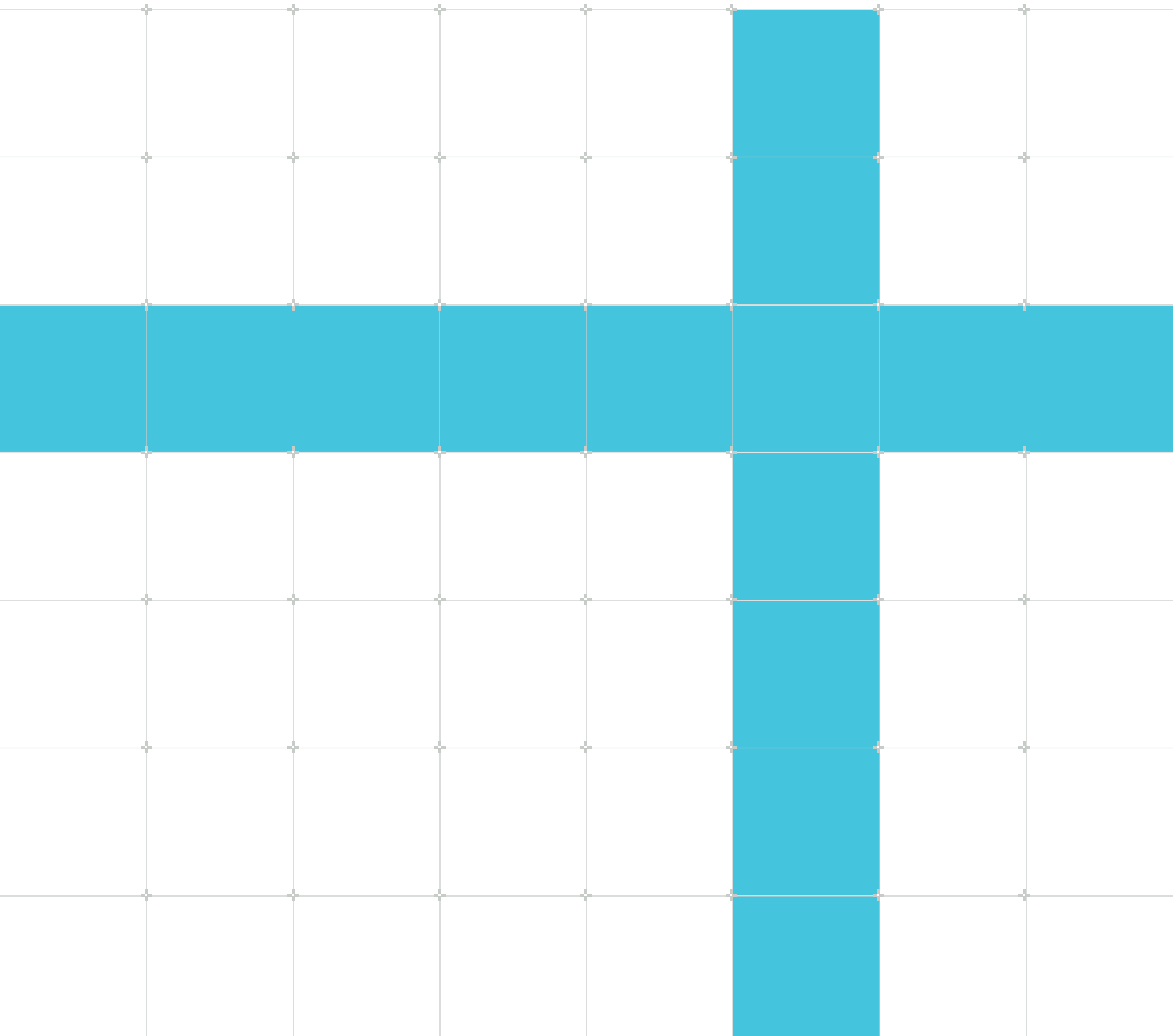
Vectorscan Porting Analysis

Version 1.0

Issue 01
109794_0100_01_en



Vectorscan Porting Analysis

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	10 May 2024	Non-Confidential	First release

Proprietary Notice

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

Inclusive language commitment

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

Vectorscan is a portable fork of Intel's Hyperscan that is optimized to be more portable, faster, and even more efficient than the original project.

This document presents an analysis of the porting effort for Hyperscan, describes the history that led to Vectorscan, and identifies challenges encountered during the porting process.

The success of porting Hyperscan should provide encouragement for projects that are still waiting to be ported from a single architecture to Arm or other architectures.

2. History of Hyperscan

Although Hyperscan is now an Intel project, originally it was a project by a company called Sensory Networks.

Intel bought Sensory Networks in 2013, as reported by iTnews, CNET, and The Sydney Morning Herald.

The Hyperscan project was led by Geoff Langdale who continued to work on the project for Intel. In 2015, Intel decided to open source Hyperscan under a BSD license.

Intel's Hyperscan was heavily optimized for SSE and AVX initially, but soon it was also optimized for AVX512, taking advantage of the new instructions in the then-new Intel CPUs. In 2019 Geoff Langdale, Xiang Wang, and other Hyperscan engineers released a whitepaper on Hyperscan's internals on USENIX. The whitepaper was very well received, describing Hyperscan's novel regular expression matchers.

Gradually Hyperscan became the de-facto solution as the most performant library for regular expression matching used in Intrusion Detection Systems (IDS) including Snort and Suricata.

Around 2020, Arm-based server CPUs including the Ampere and Ampere Altra CPUs started to emerge. Because of their extremely high core-counts, extreme power savings, and very good performance, these CPUs became very attractive solutions in the data center. To allow software such as Snort to run on Arm-based servers, Hyperscan needed to be ported to Arm.

In the summer of 2020, Arm contracted VectorCamp to port Hyperscan to Arm and contribute the port back to Hyperscan. The porting effort was submitted to Hyperscan in the following two pull requests:

- https://github.com/intel/hyperscan/pull/272
- https://github.com/intel/hyperscan/pull/287

The intent was to upstream and maintain the Arm port in the original project. However, Intel rejected these changes, so Vectorscan was created as a fork with portability as a major goal.

In 2023 Intel closed Hyperscan. The last version 5.4 is still available in GitHub, but at the time of writing no recent commits have been made.

Vectorscan is now the only open-source Hyperscan-compatible project in active development.

At the time of writing, many distributions are considering dropping the original Hyperscan packages in favor of Vectorscan because of the move to open-source licensing.

3. Vectorscan development

Since Vectorscan was forked from Hyperscan, Vectorscan has achieved the following development milestones:

- 7 releases
- 660+ commits
- Complete port to Arm ASIMD and ongoing work to further optimize for SVE and SVE2
- Fat runtime support for Arm SVE and Arm SVE2
- Support for Power8/9 VSX
- Loongson LSX port under review
- Vectorscan has been included in most major Linux distributions:
  - Official upstream Debian support
  - Ubuntu
  - SUSE
  - Fedora
  - Gentoo
  - Alpine
- Official Apple Silicon support added for MacOS on M1,M2, and M3 CPUs, available using Homebrew
- FreeBSD support for all architectures is in progress, and it will soon be a supported architecture
- Major refactoring and simplification of the source code, while still maintining the same or better performance on Intel and other architectures
- SIMDe integration to run on platforms with no officially supported SIMD engine, and to test alternative backends for existing ports
- Integration with buildbot-based CI, testing every release on more than 100 combinations of configurations and compilers
- Language bindings:
  - Rust bindings
  - Python bindings (repository)
- Tools that use Vectorscan:
  - VectorGrep
  - NoseyParker

3.1 Roadmap

Active development of Vectorscan continues, with the current roadmap of features as follows:

- 5.4.x
  - Current efforts continue to refactor the code to make it more maintainable, more readable and more portable. The goal is to have a much smaller, faster and more portable 100% Hyperscan 5.4 replacement. No extra functionality will be added during this refactoring.
  - Refactoring of the Fat runtime dispatcher to use function tables and avoid the hardware features check for each function call.
- 5.5.x
  - Refactoring will continue, adding new functionality without breaking compatibility.
  - PCRE will be upgraded to the new version, potentially breaking existing functionality
  - 32-bit x86 support will be removed.
  - Investigation for new algorithms to be added.
  - Investigation for offloading certain matchers to FPGA.
  - Investigation for rewriting certain components to new languages, for example Rust, Zig, or Carbon.
  - Port-specific:
    - Complete SVE/SVE2 optimizations for Arm
    - Investigate potential optimizations with matrix extensions including AMX, SME, and MMA
    - Investigate other SIMD engines
- 6.x
  - Refactoring will be completed at this stage.
  - Migrate the source code to C23, C++23.
  - Remove Boost dependency where possible.

4. Positive aspects of porting Hyperscan

As usual when porting a piece of software to a different architecture, there are both positive and negative aspects to the porting process. This section of the guide describes some of the positive aspects encountered while porting Hyperscan.

4.1 Thorough unit and functional tests

One of the factors that made porting much easier and predictable was the extremely thorough and well designed list of unit and functional tests.

Hyperscan provides more than 14k unit tests and more than 3.7k functional tests. Unit tests are included in the `unit-internal` generated binary, and functional tests in the `unit-hyperscan` generated binary.

These thorough unit tests meant that all porting efforts were easily tested for consistency and integrity, and caught most bugs directly. The porting process ran the tests in the following sequence:

1. First run the `unit-internal` tests.
2. When all `unit-internal` tests pass, run the `unit-hyperscan` functional tests.

Following this sequence meant that when tests failed, it was easy to locate the failure and fix the code so that the test would pass.

This made the porting effort an incremental process which helped achieve the first milestone, that of correctness. After this milestone was achieved, the porting process looked for optimizations to increase performance.

Despite the comprehensive tests, there were a few cases where a bugs were found in the code even though the tests passed. There were only a few of these bugs, and it took time to track them down and fix them, but having a unit test suite made it easier to isolate the bugs and add special cases for those failures in the unit tests themselves.

4.2 Stable API and ABI

Another positive factor encountered during the porting of Hyperscan is its stable API and ABI.

API stands for *Application Program Interface*. The API is the set of the public function interfaces that the library exports for use by other applications during compiling. A stable API is important to ensure that older applications continue to compile and run without any code changes when the library is updated.

The ABI is the *Application Binary Interface*. The ABI refers to the runtime symbols in the library. Being ABI compatible means that it is possible to replace the existing Hyperscan shared library `libhs.so` libraries in a Linux installation with the respective Vectorscan libraries, and applications continue to function as before.

4.3 Basic benchmarking tool

Hyperscan includes several tools, one of which is `hsbench`, a benchmarking tool designed to measure Hyperscan's performance. Together with sample data for the original project, `hsbench` provides useful performance metrics. These metrics were fundamental in identifying performance optimizations for Arm and other platforms.

Without this benchmarking information, performance optimization would have been much more difficult.

4.4 Using CI to build for more than 100 configurations

Vectorscan uses its own Buildbot-based CI.

Submitting a PR to the `develop` branch builds that PR on all configurations. Only if the build finishes successfully (with some exceptions due to unrelated bugs) is the PR marked ready to be merged.

This means that every PR passes through a very strict quality check, being tested on different architectures, different Operating Systems, different compilers, and even different configurations for each of those setups. So, for example, it is unlikely for an optimization on x86 to break the build on Arm, and vice versa.

Using CI is also important because it allows extra steps to be added to the build, for example static code analyzers and code formatters.

5. Challenges encountered during porting

As usual when porting a piece of software to a different architecture, there are both positive and negative aspects to the porting process. This section of the guide describes some of the challenges encountered while porting Hyperscan.

5.1 Non-modular build system

Hyperscan uses `cMake` as its build system. In general, `cMake` is a good choice but the problem is that it uses a huge monolithic `cMakeLists.txt` file with all the logic and rules in the same file.

The build system uses some `cMake` included modules, but they were primarily focused on the single x86 central architecture. This made it hard to extend or modify the build system.

As part of the porting process, the `cMake`-based build system was modularized. Now most architectural decisions are separated from the core build logic. For example, these are the contents of the `cmake` folder in both Hyperscan and Vectorscan:

Hyperscan	Vectorscan
arch.cmake	archdetect.cmake
attrib.cmake	attrib.cmake
backtrace.cmake	backtrace.cmake
boost.cmake	boost.cmake
build_wrapper.sh	build_wrapper.sh cflags-arm.cmake cflags-generic.cmake cflags-ppc64le.cmake cflags-x86.cmake compiler.cmake
config.h.in	config.h.in
formatdate.py	formatdate.py
keep.syms.in	keep.syms.in osdetection.cmake
pcre.cmake	pcre.cmake
platform.cmake	platform.cmake
ragel.cmake	ragel.cmake sanitize.cmake simde.cmake

Hyperscan	Vectorscan
`sqlite3.cmake`	`sqlite3.cmake`

Notice that all the logic for compiler flags generation, compiler, and OS detection are separated from the `CMakeLists.txt` file into their own modules.

This modularization makes it trivial to extend the build system to additional platforms, Operating Systems, and compiler configurations in the future.

Even so, the build system still needs more work. In particular the Fat Runtime build could be made modular.

5.2 Multiple warnings during compilation ignored

The original Hyperscan `CMakeLists.txt` had too many warning exceptions in order to reduce the warnings at compile time.

However this is bad practice and should be avoided. Compilers issue warnings for a reason, and problems should be fixed rather than hidden.

Some of the warnings are likely to be false positives, but it takes time and effort to identify these. To improve the situation, the porting process enabled all warnings, removed the exceptions, and added `-Werror` to make sure that the build is clean without any warnings.

The result of that effort was PR #225.

The ultimate goal is to remove the few remaining exceptions by fixing the code to eliminate these warnings.

5.3 Lack of microbenchmarking tool

The `hsbench` tool is very useful for measuring total performance. However, `hsbench` is not very useful when optimizing a single matcher. Without a suitable tool, it was not possible to assess whether attempts at optimizing a matcher would be beneficial or not.

As a result, a microbenchmarking tool `bin/benchmarks` was created. This microbenchmarking tool is not intended to act as a unit test to catch all possible cases, but it is designed to be fast and report average throughput.

The microbenchmarking tool currently supports Shufti, Truffle, Vermicelli, Noodle, and their reverse counterparts.

Running the microbenchmarking tool gives a report as follows:

```
$ ./bin/benchmarks
...
```

```
Vermicelli: no matches, 16000 * 62500 iterations, total elapsed time = 0.049 s,
 average time per call = 0.791 µs , bandwidth =  19279.390 MB/s

Vermicelli: no matches, 64000 * 15625 iterations, total elapsed time = 0.048 s,
 average time per call = 3.102 µs , bandwidth =  19675.964 MB/s

Vermicelli: no matches, 256000 * 3906 iterations, total elapsed time = 0.048 s,
 average time per call = 12.369 µs , bandwidth =  19738.642 MB/s

Vermicelli: no matches, 1024000 * 976 iterations, total elapsed time = 0.049 s,
 average time per call = 50.637 µs , bandwidth =  19285.440 MB/s

Vermicelli: no matches, 4096000 * 244 iterations, total elapsed time = 0.051 s,
 average time per call = 207.496 µs , bandwidth =  18825.673 MB/s

Vermicelli: no matches, 16384000 * 61 iterations, total elapsed time = 0.053 s,
 average time per call = 866.016 µs , bandwidth =  18042.384 MB/s

Vermicelli: no matches, 65536000 * 15 iterations, total elapsed time = 0.052 s,
 average time per call = 3444.000 µs , bandwidth =  18147.503 MB/s

Vermicelli: no matches, 262144000 * 3 iterations, total elapsed time = 0.041 s,
 average time per call = 13768.667 µs , bandwidth =  18157.168 MB/s
...
Vermicelli: 5 matches, 16000 * 62500 iterations, total elapsed time = 0.198
 s, average time per call = 1.580 µs, max bandwidth =  19325.882 MB/s, average
 bandwidth = 15444.141 MB/s
Vermicelli: 5 matches, 64000 * 15625 iterations, total elapsed time = 0.194
 s, average time per call = 6.221 µs, max bandwidth =  19624.140 MB/s, average
 bandwidth = 15697.535 MB/s
Vermicelli: 5 matches, 256000 * 3906 iterations, total elapsed time = 0.193 s,
 average time per call = 24.704 µs, max bandwidth =  19768.922 MB/s, average
 bandwidth = 15813.171 MB/s
Vermicelli: 5 matches, 1024000 * 976 iterations, total elapsed time = 0.196 s,
 average time per call = 100.395 µs, max bandwidth =  19499.284 MB/s, average
 bandwidth = 15570.167 MB/s
Vermicelli: 5 matches, 4096000 * 244 iterations, total elapsed time = 0.202 s,
 average time per call = 413.963 µs, max bandwidth =  18921.722 MB/s, average
 bandwidth = 15110.840 MB/s
Vermicelli: 5 matches, 16384000 * 61 iterations, total elapsed time = 0.207 s,
 average time per call = 1696.915 µs, max bandwidth =  18552.673 MB/s, average
 bandwidth = 14763.309 MB/s
Vermicelli: 5 matches, 65536000 * 15 iterations, total elapsed time = 0.206 s,
 average time per call = 6861.133 µs, max bandwidth =  18251.728 MB/s, average
 bandwidth = 14569.645 MB/s
Vermicelli: 5 matches, 262144000 * 3 iterations, total elapsed time = 0.165 s,
 average time per call = 27491.667 µs, max bandwidth =  18194.610 MB/s, average
 bandwidth = 14545.380 MB/s
```

This tool proved to be extremely useful when refactoring and optimizing these algorithms, as it enabled fine-tuning of the optimization.

Future plans include the following:

- Extending the microbenchmarking tool to all matchers within Vectorscan
- Possibly porting the microbenchmarking tool to a better framework, such as Google benchmark
- Add the microbenchmarking tool to CI to enable performance regression testing

5.4 Hard-coded x86 code

Because Hyperscan is an x86-only project, the codebase contains multiple assumptions about the target platform.

One of the first tasks was to separate these assumptions into separate files and provide platform-agnostic types, defines, and functions. PR #272 contains this work.

For example, x86 inline assembly code was used in algorithm code such as `src/fdr/fdr.c`. This assembly code was moved to its own file and included only when x86 architecture is detected at compile-time:

```
/* compilers don't reliably synthesize the 32-bit ANDN instruction here,
 * so we force its generation.
 */
static really_inline
u64a andn(const u32 a, const u8 *b) {
    u64a r;
#if defined(HAVE_BMI) && !defined(NO_ASM)
    __asm__ ("andn\t%2,%1,%k0" : "=r"(r) : "r"(a), "m"(*(const u32 *)b));
#else
    r = unaligned_load_u32(b) & ~a;
#endif
    return r;
}
```

This was fixed by including `util/bitutils.h` in `src/fdr/fdr.c` and using that header file as follows:

```
#if defined(ARCH_IA32) || defined(ARCH_X86_64)
#include "util/arch/x86/bitutils.h"
#endif
...
/* compilers don't reliably synthesize the 32-bit ANDN instruction here,
 * so we force its generation.
 */
static really_inline
u64a andn(const u32 a, const u8 *b) {
    return andn_impl(a, b);
}
```

At the same time, two files were defined: `util/arch/x86/bitutils.h` and `util/arch/common/bitutils.h`. Each of these files defined the `andn_impl` function:

```
util/arch/common/bitutils.h:

static really_inline
u64a andn_impl_c(const u32 a, const u8 *b) {
    return unaligned_load_u32(b) & ~a;
}
```

```
util/arch/x86/bitutils.h:

static really_inline
u64a andn_impl(const u32 a, const u8 *b) {
#if defined(HAVE_BMI) && !defined(NO_ASM)
    u64a r;
    __asm__ ("andn\t%2,%1,%k0" : "=r"(r) : "r"(a), "m"(*(const u32 *)b));
```

```
    return r;
#else
    return andn_impl_c(a, b);
#endif
}
```

This function was removed entirely because it was simpler to re-write the underlying code directly. Modern compilers are sophisticated enough to generate the correct code. Similar abstractions were implemented for the other x86-specific functions.

In addition, x86 CRC32 algorithm implementations, several x86 BMI2 bitutils functions and even SSE intrinsics were used directly in some logic code. These were also abstracted away to their own architecture-specific files and included only when compiling on x86.

Some x86-specific code remains, and future planned work will abstract this code away to their own files. This will have the benefit of reducing the amount of logic code, and also make it much easier to read and maintain.

Ideally, there should be no architecture-specific code in the algorithmic or logic code. Of course, this all should happen with no loss of performance.

This was achieved by ensuring all these functions were inlined during compilation.

5.5 SIMD abstraction is simply wrappers for x86 intrinsics

Although Hyperscan used its own SIMD intrinsics, they were just wrappers for x86 ones. Perhaps the intent was to port to other architectures in the future using this abstraction, but it did not happen.

Regardless of the intent, this abstraction only works up to a point. Because of the methods used, abstracting the SIMD C intrinsics is not enough. The vector sizes for SSE, AVX, AVX512, and AVX512VBMI are different and they use different types. There was no C++ abstract vector type that could be used for all SIMD implementations in the original Hyperscan.

Vectorscan introduces the `SuperVector` C++ class. This is essentially a container for the SIMD types for all architectures, which also provides methods for all the operations used in Vectorscan.

The future goal is to port all of the algorithms to use this class instead of the plain C SIMD intrinsics. Because this class uses partial template specialization, this allows a single C++ algorithm implementation that uses the class's methods versus the multiple implementations per SIMD engine that the original project used.

For example, see the templates refactor merged PR and the Truffle PR. In these PRs, the C versions of the Noodle, Shufti, Truffle, and Vermicelli matchers had 3 or 4 implementations each: one for SSE, AVX, AVX512, and sometimes AVX512VBMI as well. Using the `SuperVector` class reduced the code size to one third of the size for these matchers while maintaining the initial performance and even adding one more implementation for Arm Neon, and later for other architectures.

However, even with the SIMD intrinsics abstraction and the `SuperVector` class, maximum performance was not achieved. For this purpose another level of abstraction was needed.

5.6 Reliance on x86 SIMD features such as movemasks

One of the problems with the SIMD abstraction is that the original algorithms relied on features of the x86 ISA and particular instructions such as `PMOVMSKB`. Emulating this behavior on Arm and other architectures is extremely costly. The porting process had to find a solution for this problem, because it performed poorly on Arm, but removing the use of movemasks would impact x86 performance.

The solution was to abstract the part of the algorithm that used movemasks. PR-67 optimizes shift instructions, providing minimal matcher functions for each architecture. The x86 functions use the movemask-related intrinsics, but the functions for Arm and other architectures use a different method for the match, reverting to movemask emulation only at the moment where it is needed.

For example, the `firstMatch<S>` template function returns the position of a first match inside a SIMD vector, depending on a comparison mask from a search. On Arm, the function first performs a pair-wise horizontal maximum on the SIMD vector. Only if that result is non-zero does the function emulate the `PMOVMSKB` instruction. This saves multiple instructions, increasing performance and to a level comparable with x86.

```
src/util/arch/arm/match.hpp:

const u8 *firstMatch<16>(const u8 *buf, SuperVector<16> mask) {
    uint32x4_t res_t = vreinterpretq_u32_u8(mask.u.v128[0]);
    uint64_t vmax = vgetq_lane_u64 (vreinterpretq_u64_u32 (vpmaxq_u32(res_t,
 res_t)), 0);
    if (vmax != 0) {
        typename SuperVector<16>::movemask_type z = mask.movemask();
        DEBUG_PRINTF("z %08x\n", z);
        DEBUG_PRINTF("buf %p z %08x \n", buf, z);
        u32 pos = ctz32(z & 0xffff);
        DEBUG_PRINTF("match @ pos %u\n", pos);
        assert(pos < 16);
        DEBUG_PRINTF("buf + pos %p\n", buf + pos);
        return buf + pos;
    } else {
        return NULL; // no match
    }
}
```

For comparison, x86 unconditionally calculates the movemask for the vector supplied:

```
src/util/arch/x86/match.hpp:

const u8 *firstMatch<16>(const u8 *buf, SuperVector<16> v) {
    SuperVector<16>::movemask_type z = v.movemask();
    DEBUG_PRINTF("buf %p z %08x \n", buf, z);
    DEBUG_PRINTF("z %08x\n", z);
    if (unlikely(z != 0xffff)) {
        u32 pos = ctz32(~z & 0xffff);
        DEBUG_PRINTF("~z %08x\n", ~z);
        DEBUG_PRINTF("match @ pos %u\n", pos);
        assert(pos < 16);
```

```
        return buf + pos;
    } else {
        return NULL; // no match
    }
}
```

Consider the original implementation of `v.movemask()` on Arm:

```
src/util/supervector/arch/arm/impl.cpp:

template <>
really_inline typename SuperVector<16>::movemask_type
 SuperVector<16>::movemask(void) const
{
    SuperVector powers{0x8040201008040201UL};

    // Compute the mask from the input
    uint64x2_t mask  =
 vpaddlq_u32(vpaddlq_u16(vpaddlq_u8(vandq_u8((uint16x8_t)u.v128[0],
 powers.u.v128[0]))));
    uint64x2_t mask1 = (m128)vextq_s8(mask, vdupq_n_u8(0), 7);
    mask = vorrq_u8(mask, mask1);

    // Get the resulting bytes
    uint16_t output;
    vst1q_lane_u16(&output, (uint16x8_t)mask, 0);
    return static_cast<typename SuperVector<16>::movemask_type>(output);
}
```

Simply emulating the x86 intrinsics directly would not produce optimal performance.

External contributions by SIMD experts including Danila Kutenin from Google increased performance significantly. For example, PR-113 increases performance by optimizing the movemask emulation on non-x86 implementations using shifts. For more information, see Danila's blog post on Arm Community: Bit twiddling with Arm Neon: beating SSE movemasks, counting bits and more.

The current implementation of the `v.movemask()` function on Arm and x86 is as follows:

```
src/util/arch/arm/match.hpp:

template <>
really_really_inline
const u8 *first_non_zero_match<16>(const u8 *buf, SuperVector<16> mask, u16 const
 UNUSED len) {
    uint32x4_t m = mask.u.u32x4[0];
    uint64_t vmax = vgetq_lane_u64 (vreinterpretq_u64_u32 (vpmaxq_u32(m, m)), 0);
    if (vmax != 0) {
        typename SuperVector<16>::comparemask_type z = mask.comparemask();
        DEBUG_PRINTF("z %08llx\n", z);
        DEBUG_PRINTF("buf %p z %08llx \n", buf, z);
        u32 pos = ctz64(z) / SuperVector<16>::mask_width();
        DEBUG_PRINTF("match @ pos %u\n", pos);
        assert(pos < 16);
        DEBUG_PRINTF("buf + pos %p\n", buf + (pos));
        return buf + pos;
    } else {
        return NULL; // no match
    }
```

```
}
```

```
src/util/arch/x86/match.hpp:

template <>
really_really_inline
const u8 *first_non_zero_match<16>(const u8 *buf, SuperVector<16> v, u16 const
 UNUSED len) {
    assert(SuperVector<16>::mask_width() == 1);
    SuperVector<16>::comparemask_type z = v.comparemask();
    DEBUG_PRINTF("buf %p z %08llx \n", buf, z);
    DEBUG_PRINTF("z %08llx\n", z);
    if (unlikely(z)) {
        u32 pos = ctz32(z);
        DEBUG_PRINTF("~z %08llx\n", ~z);
        DEBUG_PRINTF("match @ pos %u\n", pos);
        assert(pos < 16);
        return buf + pos;
    } else {
        return NULL; // no match
    }
}
```

This is the `comparemask()` implementation for both Arm and x86:

```
src/util/supervector/arch/arm/impl.cpp:

template <>
really_inline typename SuperVector<16>::comparemask_type
SuperVector<16>::comparemask(void) const {
    return static_cast<typename SuperVector<16>::comparemask_type>(
        vget_lane_u64((uint64x1_t)vshrn_n_u16(u.u16x8[0], 4), 0));
}
```

```
src/util/supervector/arch/x86/impl.cpp:

template <>
really_inline typename SuperVector<16>::comparemask_type
SuperVector<16>::comparemask(void) const {
    return (u32)_mm_movemask_epi8(u.v128[0]);
}
```

There is ongoing work to continue this abstraction of x86 features to make the code more performant on other architectures, while maintaining the same or better performance on x86.

5.7 SIMD Everywhere

An additional aim of the porting process was to investigate whether Vectorscan could run on non-SIMD architectures, or on architectures that have a SIMD unit that is not yet supported. Of course performance would not be as high without SIMD, but this effort would still be beneficial for those architectures and help Vectorscan increase its userbase.

One initial idea was to write a new SIMD emulation layer, but early investigation suggested that the SIMD Everywhere (SIMDe) library would be suitable instead. PR-203 adds a port for the SIMDe library.

The SIMDe port also provided the opportunity to measure the performance of the SIMDe emulation and compare it to the Arm implementation as reference. As expected, the Arm-specific implementation performs better, but the SIMDe port is useful for other architectures.

5.8 Old-style C and C++ code

The current implementations are a mix of old-style C and C++ code. In several places, unfortunately sometimes in critical code paths, the heavy use of C macros is apparent.

Consider the following code in `src/fdr/fdr.c`:

```
#define FDR_MAIN_LOOP(zz, s, get_conf_fn)                                      \
    do {                                                                       \
        const u8 *tryFloodDetect = zz->floodPtr;                               \
        const u8 *start_ptr = zz->start;                                       \
        const u8 *end_ptr = zz->end;                                           \
        for (const u8 *itPtr = ROUNDDOWN_PTR(start_ptr, 64); itPtr + 4*ITER_BYTES <=
 end_ptr;      \
             itPtr += 4*ITER_BYTES) {                                          \
             __builtin_prefetch(itPtr);                                        \
        }                                                                      \
                                                                               \
        for (const u8 *itPtr = start_ptr; itPtr + ITER_BYTES <= end_ptr;       \
             itPtr += ITER_BYTES) {                                            \
             if (unlikely(itPtr > tryFloodDetect)) {                           \
                 tryFloodDetect = floodDetect(fdr, a, &itPtr, tryFloodDetect,\
                                              &floodBackoff, &control,         \
                                              ITER_BYTES);                     \
                 if (unlikely(control == HWLM_TERMINATE_MATCHING)) {           \
                     return HWLM_TERMINATED;                                   \
                 }                                                             \
             }                                                                 \
             __builtin_prefetch(itPtr + ITER_BYTES);                           \
             u64a conf0;                                                       \
             u64a conf8;                                                       \
             get_conf_fn(itPtr, start_ptr, end_ptr, domain_mask_flipped,       \
                         ft, &conf0, &conf8, &s);                              \
             do_confirm_fdr(&conf0, 0, &control, confBase, a, itPtr,           \
                            &last_match_id, zz);                               \
             do_confirm_fdr(&conf8, 8, &control, confBase, a, itPtr,           \
                            &last_match_id, zz);                               \
             if (unlikely(control == HWLM_TERMINATE_MATCHING)) {               \
                 return HWLM_TERMINATED;                                       \
             }                                                                 \
        } /* end for loop */                                                   \
    } while (0)                                                                \
```

Also consider how the macro is called, also in `src/fdr/fdr.c`:

```
        switch (stride) {
        case 1:
            FDR_MAIN_LOOP(z, state, get_conf_stride_1);
            break;
        case 2:
            FDR_MAIN_LOOP(z, state, get_conf_stride_2);
            break;
        case 4:
            FDR_MAIN_LOOP(z, state, get_conf_stride_4);
            break;
        default:
```

```
            break;
        }
```

This code is very difficult to debug, let alone optimize. In the case of a segmentation fault, the debugger shows the macro as a single line, and the only way to diagnose the problem is by looking at the disassembly. There is ongoing work to replace these macros in FDR with normal inline functions and similar work will be done for instances in the code, for example Sheng, Limex, Teddy, and so on.

Additionally, multiple fixes were made in the C++ code to ensure that it compiles to the C++17 specification and eliminate all warnings.

This is not just a matter of taste. The use of old-style C and hard-coded offsets was the reason for Issue 184, a hard-to-diagnose segmentation fault when using SVE2 on Arm Fat runtimes. Upgrading to the newer C specification results in safer and more robust code.

As a long term goal, the roadmap for Vectorscan 6 is to move to C23 and C++23 only. Static code analyzers will be added to the CI to monitor and control this migration.

5.9 No static analysis or syntax formatting

Manually migrating source code to newer C and C++ specs is difficult. Because of this, the porting process added static code analyzers including `cppcheck` and `clang-tidy` to the CI, to help with migration and improvement of the code in general.

The future roadmap plans to enforce code-formatting guidelines to keep a consistent code style. Although this is not important for the compilers, it is important for the human developers that read the code.

5.10 Memory allocations in critical paths

Especially in the C++ code, some critical paths perform dynamic memory allocation. It is likely that these dynamic memory allocations impact performance. The future roadmap plans to perform a thorough analysis using specialized tools to detect these allocations and if possible eliminate them.

5.11 Improve documentation and comments in the code

The existing external documentation describes high-level concepts in detail, but does not describe the internals of the algorithms.

The future roadmap plans to improve this documentation, and also add the missing information as comments in the source code.