hyperscan测试文档

编译

按照官方提供的编译文档，安装 ragel,boost,pcre,sqlite 等相关工具即可，需要注意的如果使用开源的 vectorscan 来编译的话，需要升级 g++ 到 **9.0** 版本以上。

- vectorscan 作为 hyperscan 的开源社区的另一个版本，目前已经支持 arm，powerpc 平台，并且也已经实现了纯软件的版本，使用了 C++ 完全实现和模拟了 SIMD 指令，并且已经支持了 mips 的 vector 指令集，可以在 ios 平台下运行，并且已有相关开源社区的支持。目前平台地址为：vectorscan
  - 该分支支持多种平台，并且有社区相关技术人员支持与持续开发集成，由于 hyperscan 不支持 arm 平台，社区人员自行做了适配，并将该分支提交给 intel ，但是被 hyperscan 官方给拒绝了，社区人员就开辟了此分支专门支持 arm 平台。
  - 该分支的优点是适配多种平台，缺点是对 hyperscan 本身的框架改动非常大，将所有关于关于跨平台的操作全部拿出来单独形成一个模块做适配，另一方面该分支使用的 C++ 17 来开发的，开发难度还是比纯 C 高一些，并且对其中的 cmake 文件改动非常大，需要仔细分析和阅读。
  - 后期期许以该版本为实例来持续的演进和适配，但需要的开发技巧和技术能力较高。
- 目前主要的工作还是基于 hyperscan 提供的官方的版本来做修改和测试。目前主要基于 5.2.0 的版本，也即华为提供的鲲鹏平台的 hyperscan 来做修改和测试，目前修改的版本已经成功可以在华为鲲鹏平台、x86、飞腾下面运行，华为的官方地址为hyperscan。当前的代码平台托管在码云上，项目地址:proj

测试方法

目前测试主要基于 hyperscan 的内部提供的模块测试用例和 hyperscan 提供基于正则测试数据集合来进行测试，主要利用 hyperscan 提供的四个工具： unit-internal ， unit-chimera ， hsbench , hscollider 。

1. unit-internal

hyperscan 自带的测试用例测试：目前 hyperscan 通过修改 cmake 文件来制定编译模式，在非 release 模式下会生成单元测试程序，可以在 build 的 bin 目录下多了两个单元测试程序 unit-internal 与 unit-cinema ， unit-internal 主要为 hyperscan 内部各个模块的单元测试，修改程序后能够通过 unit-internal 的测试用例，基本上能够保证程序的正确性。 unit-internal 的所有测试用例都是利用 googletest 框架编写的测试用例，在执行扩展相关模块和程序时，可以自行的扩展相关的测试用例。

2. uint-chimera

unit-chimera 主要为了支持 pcre 的全部语法，而与 pcre 进行混合的一个工具。如果使用 chimera 进行编译时，首先会将规则进行分配，将 hyperscan 不支持的规则过滤出来使用 pcre 进行编译，同时在数据库中分别存储两种规则的编译结果，匹配时是首先会调用 hyperscan 扫描它支持的规则，然后再利用 pcre 将 hyperscan 不支持的规则再扫描一遍，从而保证其能够完全兼容 pcre 的语法规则，但是这种使用方法耗费资源较大，性能较低。

3. hsbench

hsbench 为 hypersan 官方提供的性能测试工具，可以支持对规则文件进行编译并匹配提供样本文件。

- 规则文件格式:

```
<id>:/<regex>/<flags>
```

- 语料库文件必须以 SQLite 数据库的形式提供， hyperscan 提供有脚本工具可以将 pcap 文件和普通文本文件转化为 db 文件。
- 测试的规则库主要利用 snort_literals ， snort_pcres , teakettle_2500 三个规则库：
  - snort_literals：这是从Snort 3网络入侵检测系统自带的样本规则集中提取的一组3316个纯字符串规则，可在https://github.com/snortadmin/snort3查看。其中一些标记有HS_FLAG_CASELESS标志，以便它们不区分大小写进行匹配，并且都使用HS_FLAG_SINGLEMATCH来限制每个模式的每次扫描只匹配一次。
  - snort_pcres :这是从包含 Snort 3 的样本规则集中提取的一组847个正则表达式集，取自针对 HTTP 流量的规则。值得注意的是，这些只是从规则的“pcre：”选项中提取的规则，并且 Hyperscan 的匹配在语义上和 Snort 当中的匹配会有所不同。这个样本旨在展示 Hyperscan 在同一时间匹配一个正则表达式集的能力。
  - teakettle_2500：这是由一个脚本生成的2500个合成的有限复杂度规则集，它们主要由字典单词组成，并在单词间插入重复的字符集和分隔符。
- 测试的样本库库主要利用 alexa200 ， gutenberg ， test ， 这两个样本库。
  - gutenberg.db：来自 Gutenberg 的英文文本集合，我们将其分成多条大小为 10240 字节数据流，每条流包含多个 2048 字节的数据块。

- alexa200.db：通过浏览 Alexa 上列出的常用网站以 PCAP 形式捕获的大规模流量。该文件包含 130957 个块（对应于原始报文），且仅包括发送至或来自 80 端口的流量。
- news.db：这个样本数据为自己在网上下载的英文新闻生成的数据测试集，大概有45mb，类似的数据库可以到自然语言处理相关的数据库进行下载，然后利用hyperscan提供的脚本工具制作数据库样本。

- 性能测试工具使用说明如下：

```
Usage: hsbench [OPTIONS...]

Options:

  -h              Display help and exit.
  -G OVERRIDES    Overrides for the grey box.
  -e PATH         Path to expression directory.
  -s FILE         Signature file to use.
  -z NUM          Signature ID to use.
  -c FILE         File to use as corpus.
  -n NUMBER       Repeat scan NUMBER times (default 20).
  -N              Benchmark in block mode (default: streaming).
  -V              Benchmark in vectored mode (default: streaming).
  -H              Benchmark using Chimera (if supported).
  -P              Benchmark using PCRE (if supported).
  -T CPU,CPU,... or -T CPU-CPU
                  Benchmark with threads on specified CPUs or CPU range.
  -i DIR          Don't compile, load from files in DIR instead.
  -w DIR          After compiling, save to files in DIR.
  -d NUMBER       Set SOM precision mode (default: 8 (large)).
  -E DISTANCE     Match all patterns within edit distance DISTANCE.

  --per-scan      Display per-scan Mbit/sec results.
  --echo-matches  Display all matches that occur during scan.
  --sql-out FILE  Output sqlite db.
  --literal-on    Use Hyperscan pure literal matching.
  -S NAME         Signature set name (for sqlite db).
```

常用命令如下:

```
hsbench -e [规则文件路径] -c [样本文件] -n [扫描次数] -T [指令的线程数] -[(N,V),block模式还是vector模式] -[(H,P), chiemrea混合模式，PCRE模式]
```

运行完成后，会输出性能测试的时间以及吞吐量。

4. hscollider

hyperscan 官方提供的测试结果正确性比对工具，会使用 pcre 对规则进行匹配，匹配的结果与 hyperscan 检测的结果进行比对，如果完全匹配则说明 hyperscan 的检测结果的正确性，该功能较为简单。

- 该工具使用说明如下：

```
Usage: ./hscollider [OPTIONS...]

General Options:

  -h              Display help and exit.
  -G OVERRIDES    Overrides for the grey box.
  -e PATH         Path to expression directory or file.
  -s FILE         Signature file to use.
  -z NUM          Signature ID to use.
  -c FILE         Load corpora from FILE rather than using generator.
  -w FILE         After running, save corpora (with matches) to FILE.
  -a [BAND]       Compile all expressions in UE2 (but still match singly).
                  If BAND, compile patterns in groups of size BAND.
  -t NUM          Use streaming mode, split data into ~NUM blocks.
  -V NUM          Use vectored mode, split data into ~NUM blocks.
  -H              Use hybrid mode.
  -Z {R or 0-63}  Only test one alignment, either as given or 'R' for random.
  -q              Quiet; display only match differences, no other failures.
  -v              Verbose; display successes as well as failures.

Pattern flags:

  -8              Force UTF8 mode on all patterns.
  -L              Apply HS_FLAG_SOM_LEFTMOST to all patterns.
  -E DISTANCE     Match all patterns within edit distance DISTANCE.
  --prefilter     Apply HS_FLAG_PREFILTER to all patterns.
  --no-groups     Disable capturing in Hybrid mode.

Testing mode options:

  -d NUM          Set SOM precision mode (default: 8 (large)).
  -O NUM          In streaming mode, set initial offset to NUM.
  -k NUM          Terminate callback after NUM matches per pattern.
  --copy-scratch  Copy scratch after each scan call.
  --copy-stream   Copy stream state after each scan call.
  --compress-expand Compress and expand stream state after each scan call.
  --compress-reset-expand Compress, reset and expand stream state after each scan call.
  --mangle-scratch Mangle scratch space after each scan call.
  --no-nfa        Disable NFA graph execution engine.
  --no-pcre       Disable PCRE engine.
  --test-nfa      Disable UE2 engine (test NFA against PCRE).
  --abort-on-fail Abort, rather than exit, on failure.
  --no-signal-handler Do not handle handle signals (to generate backtraces).
  --literal-on    Use Hyperscan pure literal matching.

Memory and resource control options:

  -T NUM          Run with NUM threads.
  -M NUM          Set maximum memory allocated to NUM megabytes per thread.
                  (0 means no limit, default is 1000 MB).
  -m NUM          Set PCRE_MATCH_LIMIT (default: 10000000).
  -r NUM          Set PCRE_MATCH_LIMIT_RECURSION (default: 10000).

Cross-compiling:

  -x NAME         Cross-compile for arch NAME.
  -i DIR          Don't compile, load from files in DIR instead.
  -o DIR          After compiling, save to files in DIR.

Corpus generation options:

  -n NUM          Max corpora to generate for a given signature (default: 500000).
  -R NUM          Random seed to use (default: seeded from time()).
  -p NUM,NUM,NUM  Percentage probabilities of (match,unmatch,random) char.
  -C NUM,NUM      Follow cycles (min,max) times.
  -P NUM,NUM      Add a random prefix of length between (min,max).
  -S NUM,NUM      Add a random suffix of length between (min,max).
  -D NUM          Apply an edit distance (default: 0) to each corpus.
  -b NUM          Limit alphabet to NUM characters, starting at lower-case 'a'.
```

- 该工具常用的使用命令如下：

```
hscollider -e [规则文件] -c [样本文件] -Z [对齐字节] -T [线程数目] -vv
```

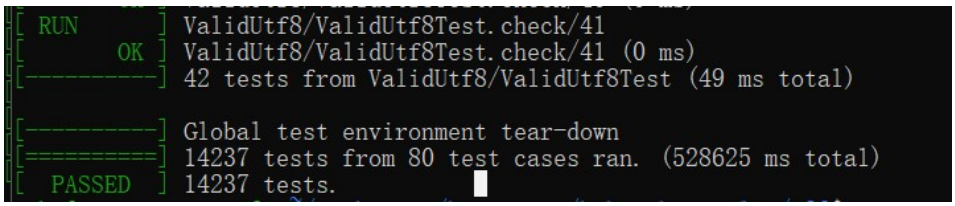
测试结果

测试结果主要分为性能测试和功能性测试，性能测试主要对比利用 simd 指令加速和非 simd 指令加速之间的对比，功能性测试主要测试 hyperscan 的检测结果是否正确，目前通过与 pcre 的检测结果进行对比。实际对比可以发现，在使用 hyperscan 的内部测试用例测试上，使用 simd 指令加速与不使用 simd 加速二者几乎没有大的性能差别，但是在进行大规则的匹配与查找时，使用 simd 指令加速后性能会得到极大的改善，使用 simd 加速后吞吐量达到使用纯软件的加速 3 到 4 倍。

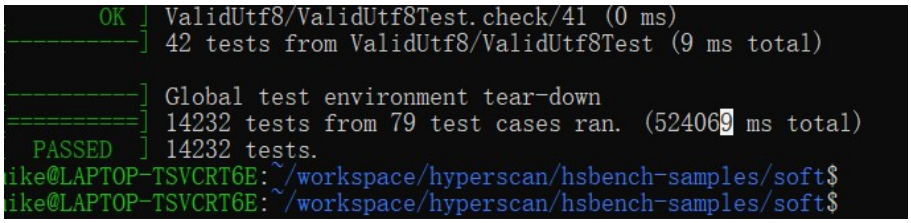
性能测试：

1. hyperscan 自带的测试用例的测试结果：

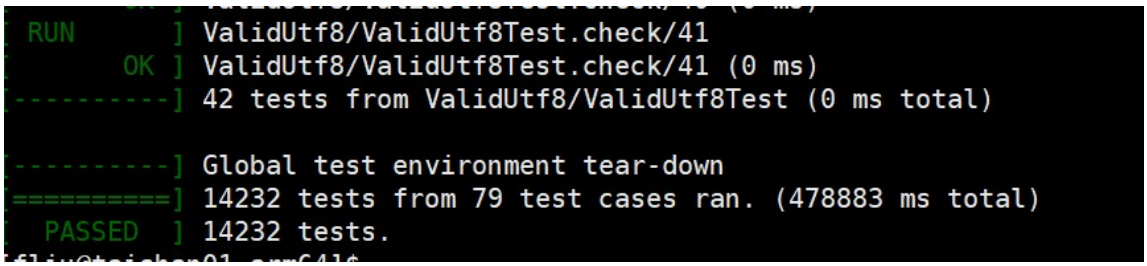
- unit-internal 测试：
  unit 自带的测试用例总计约为 14240 个，所有的测试用例
- X86 平台使用 simd 指令集：



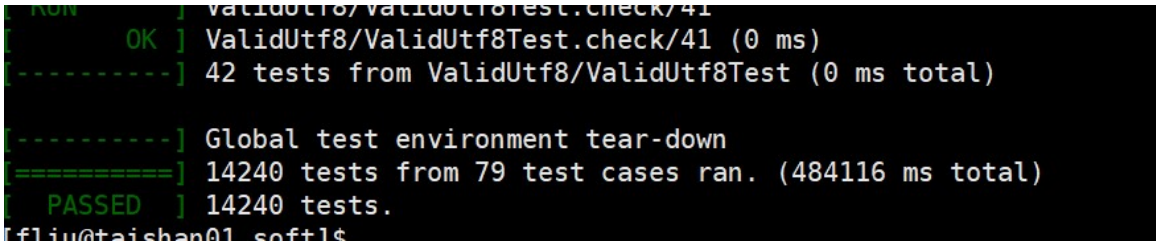
- X86 平台不使用 simd 指令集：



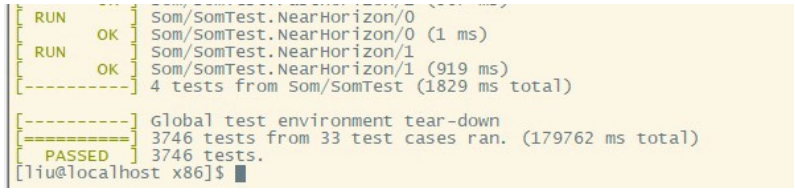
- 华为鲲鹏平台使用 simd 指令集：



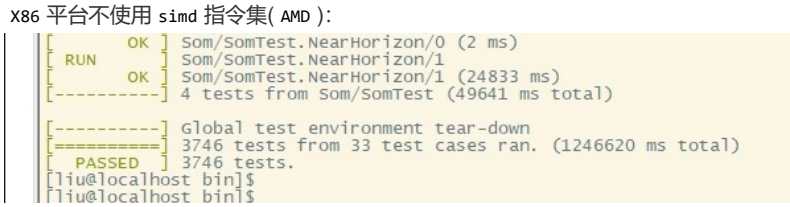
- 华为鲲鹏平台不使用 simd 指令集：



- X86 平台使用 simd 指令集(AMD)：



- X86 平台不使用 simd 指令集(AMD)：



平台名称	是否使用SIMD	CPU	内存	操作系统	运行时间

平台名称	是否使用SIMD	CPU	内存	操作系统	运行时间
intel X86	是	intel core-i58250u， 总计8核心	8G	ubuntu	528250ms
intel X86	否	intel core-i58250u， 总计8核心	8G	ubuntu	524069ms
华为鲲鹏	是	鲲鹏 arm64，总计96核心	128G	Centos	478883ms
华为鲲鹏	否	鲲鹏 arm64，总计96核心	128G	Centos	484116ms
amd X86	是	AMD 7351 16-core X 4， 总计64核心	128G	ubuntu	179762ms
amd X86	否	AMD 7351 16-core X 4，总计64核心	128G	ubuntu	1246620ms

1. 正则测试集测试结果：

   语料库分别使用 `alexa200`， `gurage`，`news`，正则规则库分别使用 `snort_literals`，`snort_pcres`， `teakettle_2500`。分别使用 `block` 模式 和 `stream` 模式。

- `block` 模式下使用 `snort_literals` 正则库匹配三种语料库对比测试结果如下：
  - `alexa200` 语料库测试：
  - `gurage` 数据集测试：
  - `news` 数据测试：
  - `CPU` 参数设置为 `10` 和 `1`

plateform	corpora	pcre	mode	CPU	Max throughput	Mean throughput	Time spent	remark
X86 simd	alexa200	snort_literals	block	10	3879.81 Mbit/sec	3271.73 Mbit/sec	8.660 seconds	intel
X86 common	alexa200	snort_literals	block	10	1245.14 Mbit/sec	1007.11 Mbit/sec	28.134 seconds	intel
arm simd	alexa200	snort_literals	block	10	968.67 Mbit/sec	967.40 Mbit/sec	29.289 seconds	arm64
arm common	alexa200	snort_literals	block	10	1,383.96 Mbit/sec	746.05 Mbit/sec	37.979 seconds	arm64
X64 simd (amd)	alexa200	snort_literals	block	10	3,739.79 Mbit/sec	3,604.77 Mbit/sec	7.860 seconds	amd
X64 common (amd)	alexa200	snort_literals	block	10	1,437.68 Mbit/sec	1,405.80 Mbit/sec	20.155 seconds	amd
arm simd	alexa200	snort_literals	block	1	967.28 Mbit/sec	524.08 Mbit/sec	54.064 seconds	arm64
arm common	alexa200	snort_literals	block	1	690.26 Mbit/sec	687.33 Mbit/sec	41.223 seconds	arm64
X64 simd (amd)	alexa200	snort_literals	block	1	3,756.10 Mbit/sec	3,651.91 Mbit/sec	7.759 seconds	amd
X64 common (amd)	alexa200	snort_literals	block	1	721.01 Mbit/sec	712.55 Mbit/sec	39.764 seconds	amd
X86 simd	gurage	snort_literals	block	10	4029.79 Mbit/sec	3544.20 Mbit/sec	0.303 seconds	intel
X86 common	gurage	snort_literals	block	10	1013.83 Mbit/sec	786.77 Mbit/sec	1.363 seconds	intel
arm simd	gurage	snort_literals	block	10	931.19 Mbit/sec	930.52 Mbit/sec	1.152 seconds	arm64
arm common	gurage	snort_literals	block	10	740.65 Mbit/sec	666.33 Mbit/sec	1.609 seconds	arm64
X64 simd (amd)	gurage	snort_literals	block	10	4,504.96 Mbit/sec	3,823.75 Mbit/sec	0.280 seconds	amd
X86 common (amd)	gurage	snort_literals	block	10	1,363.05 Mbit/sec	1,323.28 Mbit/sec	0.810 seconds	amd
arm simd	gurage	snort_literals	block	1	507.51 Mbit/sec	463.44 Mbit/sec	2.313 seconds	arm64
arm common	gurage	snort_literals	block	1	739.92 Mbit/sec	662.36 Mbit/sec	1.619 seconds	arm64
X86 simd (amd)	gurage	snort_literals	block	1	4,532.78 Mbit/sec	3,743.07 Mbit/sec	0.286 seconds	amd
X86 common (amd)	gurage	snort_literals	block	1	738.69 Mbit/sec	677.49 Mbit/sec	1.583 seconds	amd
X86 simd	news	snort_literals	block	10	1275.71 Mbit/sec	1080.36 Mbit/sec	6.658 seconds	intel
X86 common	news	snort_literals	block	10	485.84 Mbit/sec	336.36 Mbit/sec	21.384 seconds	intel
arm simd	news	snort_literals	block	10	500.30 Mbit/sec	274.09 Mbit/sec	26.242 seconds	arm64
arm common	news	snort_literals	block	10	295.10 Mbit/sec	294.00 Mbit/sec	24.465 seconds	arm64
X64 simd (amd)	news	snort_literals	block	10	1,450.39 Mbit/sec	1,414.54 Mbit/sec	5.085 seconds	amd
X86 common (amd)	news	snort_literals	block	10	615.10 Mbit/sec	608.37 Mbit/sec	11.823 seconds	amd
arm simd	news	snort_literals	block	1	249.62 Mbit/sec	249.07 Mbit/sec	28.878 seconds	arm64
arm common	news	snort_literals	block	1	296.23 Mbit/sec	294.59 Mbit/sec	24.416 seconds	arm64
X86 simd (amd)	news	snort_literals	block	1	1,452.21 Mbit/sec	1,439.92 Mbit/sec	4.995 seconds	amd
X86 common (amd)	news	snort_literals	block	1	311.58 Mbit/sec	305.66 Mbit/sec	23.532 seconds	amd

- `stream` 模式下使用 `snort_pcres` 正则库匹配三种语料库对比
  - `alexa200` 语料库测试：
  - `gurage` 数据集测试：
  - `news` 数据测试：
  - `CPU` 参数设置为 `10` 和 `1`

plateform	corpora	pcre	mode	CPU	Max throughput	Mean throughput	Time spent	remark
X86 simd	alexa200	snort_pcres	stream	10	478.22 Mbit/sec	400.85 Mbit/sec	70.685 seconds	intel
X86 common	alexa200	snort_pcres	stream	10	170.58 Mbit/sec	149.30 Mbit/sec	189.781 seconds	intel
arm simd	alexa200	snort_pcres	stream	10	115.16 Mbit/sec	100.69 Mbit/sec	281.400 seconds	arm64
arm common	alexa200	snort_pcres	stream	10	130.23 Mbit/sec	106.23 Mbit/sec	266.716 seconds	arm64
X64 simd (amd)	alexa200	snort_pcres	stream	10	420.82 Mbit/sec	408.64 Mbit/sec	69.337 seconds	amd
X86 common (amd)	alexa200	snort_pcres	stream	10	155.19 Mbit/sec	154.60 Mbit/sec	183.271 seconds	amd
arm simd	alexa200	snort_pcres	stream	1	99.61 Mbit/sec	99.43 Mbit/sec	284.951 seconds	arm64
arm common	alexa200	snort_pcres	stream	1	209.43 Mbit/sec	106.85 Mbit/sec	265.180 seconds	arm64
X86 simd (amd)	alexa200	snort_pcres	stream	1	416.81 Mbit/sec	254.57 Mbit/sec	111.302 seconds	amd
X86 common (amd)	alexa200	snort_pcres	stream	1	110.07 Mbit/sec	80.25 Mbit/sec	353.060 seconds	amd
X86 simd	gurage	snort_pcres	stream	10	727.08 Mbit/sec	718.82 Mbit/sec	1.492 seconds	intel
X86 common	gurage	snort_pcres	stream	10	179.12 Mbit/sec	144.62 Mbit/sec	7.414 seconds	intel
arm simd	gurage	snort_pcres	stream	10	103.44 Mbit/sec	102.12 Mbit/sec	10.499 seconds	arm64
arm common	gurage	snort_pcres	stream	10	109.57 Mbit/sec	107.39 Mbit/sec	9.984 seconds	arm64
X64 simd (amd)	gurage	snort_pcres	stream	10	528.65 Mbit/sec	507.95 Mbit/sec	2.111 seconds	amd
X86 common (amd)	gurage	snort_pcres	stream	10	181.72 Mbit/sec	178.17 Mbit/sec	6.018 seconds	amd
arm simd	gurage	snort_pcres	stream	1	103.54 Mbit/sec	102.43 Mbit/sec	10.467 seconds	arm64

arm common	gurage	snort_pcres	stream	1	217.22 Mbit/sec	216.96 Mbit/sec	4.942 seconds	arm64
X86 simd (amd)	gurage	snort_pcres	stream	1	275.10 Mbit/sec	264.33 Mbit/sec	4.056 seconds	amd
X86 common (amd)	gurage	snort_pcres	stream	1	93.00 Mbit/sec	90.45 Mbit/sec	11.854 seconds	amd
X86 simd	news	snort_pcres	stream	10	183.70 Mbit/sec	157.81 Mbit/sec	45.578 seconds	intel
X86 common	news	snort_pcres	stream	10	56.99 Mbit/sec	50.02 Mbit/sec	143.791 seconds	intel
arm simd	news	snort_pcres	stream	10	42.79 Mbit/sec	31.64 Mbit/sec	227.321 seconds	arm64
arm common	news	snort_pcres	stream	10	42.08 Mbit/sec	31.47 Mbit/sec	228.530 seconds	arm64
X64 simd (amd)	news	snort_pcres	stream	10	119.24 Mbit/sec	118.47 Mbit/sec	60.712 seconds	amd
X86 common (amd)	news	snort_pcres	stream	10	50.54 Mbit/sec	50.32 Mbit/sec	142.925 seconds	amd
arm simd	news	snort_pcres	stream	1	62.17 Mbit/sec	53.35 Mbit/sec	134.812 seconds	arm64
arm common	news	snort_pcres	stream	1	62.72 Mbit/sec	62.67 Mbit/sec	114.762 seconds	arm64
X86 simd (amd)	news	snort_pcres	stream	1	59.92 Mbit/sec	59.73 Mbit/sec	120.416 seconds	amd
X86 common (amd)	news	snort_pcres	stream	1	50.57 Mbit/sec	39.00 Mbit/sec	184.405 seconds	amd

- `stream` 模式下使用 `teakettle_2500` 正则库匹配三种语料库对比
  - `alexa200` 语料库测试：
  - `gurage` 数据集测试：
  - `news` 数据测试：
  - `CPU` 参数设置为 `10` 和 `1`

plateform	corpora	pcre	mode	CPU	Max throughput	Mean throughput	Time spent	remark
X86 simd	alexa200	teakettle_2500	stream	10	5290.99 Mbit/sec	4710.86 Mbit/sec	6.015 seconds	intel
X86 common	alexa200	teakettle_2500	stream	10	2029.69 Mbit/sec	1891.79 Mbit/sec	14.977 seconds	intel
arm simd	alexa200	teakettle_2500	stream	10	1,191.06 Mbit/sec	634.05 Mbit/sec	44.688 seconds	arm64
arm common	alexa200	teakettle_2500	stream	10	1,872.09 Mbit/sec	1,138.16 Mbit/sec	24.895 seconds	arm64
X64 simd (amd)	alexa200	teakettle_2500	stream	10	4,895.57 Mbit/sec	4,727.09 Mbit/sec	5.994 seconds	amd
X86 common (amd)	alexa200	teakettle_2500	stream	10	1,835.13 Mbit/sec	1,784.37 Mbit/sec	15.879 seconds	amd
arm simd	alexa200	teakettle_2500	stream	1	1,192.05 Mbit/sec	1,190.04 Mbit/sec	23.809 seconds	arm64
arm common	alexa200	teakettle_2500	stream	1	1,874.98 Mbit/sec	1,867.45 Mbit/sec	15.173 seconds	arm64
X86 simd (amd)	alexa200	teakettle_2500	stream	1	2,491.50 Mbit/sec	2,407.71 Mbit/sec	11.768 seconds	amd
X86 common (amd)	alexa200	teakettle_2500	stream	1	1,834.96 Mbit/sec	1,816.74 Mbit/sec	15.596 seconds	amd
X86 simd	gurage	teakettle_2500	stream	10	2633.67 Mbit/sec	2461.45 Mbit/sec	0.436 seconds	intel
X86 common	gurage	teakettle_2500	stream	10	1654.12 Mbit/sec	1351.69 Mbit/sec	0.793 seconds	intel
arm simd	gurage	teakettle_2500	stream	10	552.56 Mbit/sec	542.14 Mbit/sec	1.978 seconds	arm64
arm common	gurage	teakettle_2500	stream	10	828.32 Mbit/sec	813.47 Mbit/sec	1.318 seconds	arm64
X64 simd (amd)	gurage	teakettle_2500	stream	10	2,635.12 Mbit/sec	2,295.69 Mbit/sec	0.467 seconds	amd
X86 common (amd)	gurage	teakettle_2500	stream	10	1,693.30 Mbit/sec	1,545.17 Mbit/sec	0.694 seconds	amd
arm simd	gurage	teakettle_2500	stream	1	1,104.99 Mbit/sec	1,104.00 Mbit/sec	0.971 seconds	arm64
arm common	gurage	teakettle_2500	stream	1	1,663.33 Mbit/sec	1,658.10 Mbit/sec	0.647 seconds	arm64
X86 simd (amd)	gurage	teakettle_2500	stream	1	1,327.50 Mbit/sec	1,325.80 Mbit/sec	0.809 seconds	amd
X86 common (amd)	gurage	teakettle_2500	stream	1	1,704.68 Mbit/sec	1,683.06 Mbit/sec	0.637 seconds	amd
X86 simd	news	teakettle_2500	stream	10	65.87 Mbit/sec	63.49 Mbit/sec	113.284 seconds	intel
X86 common	news	teakettle_2500	stream	10	26.57 Mbit/sec	23.90 Mbit/sec	300.998 seconds	intel
arm simd	news	teakettle_2500	stream	10	18.36 Mbit/sec	14.37 Mbit/sec	500.602 seconds	arm64
arm common	news	teakettle_2500	stream	10	19.86 Mbit/sec	14.37 Mbit/sec	500.440 seconds	arm64
X64 simd (amd)	news	teakettle_2500	stream	10	50.22 Mbit/sec	50.04 Mbit/sec	143.733 seconds	amd
X86 common (amd)	news	teakettle_2500	stream	10	22.31 Mbit/sec	22.26 Mbit/sec	323.178 seconds	amd
arm simd	news	teakettle_2500	stream	1	28.65 Mbit/sec	17.32 Mbit/sec	415.360 seconds	arm64
arm common	news	teakettle_2500	stream	1	28.53 Mbit/sec	28.51 Mbit/sec	252.316 seconds	arm64
X86 simd (amd)	news	teakettle_2500	stream	1	30.80 Mbit/sec	25.62 Mbit/sec	280.788 seconds	amd
X86 common (amd)	news	teakettle_2500	stream	1	22.33 Mbit/sec	22.31 Mbit/sec	322.456 seconds	amd

经过测试对比可以发现，在 `X86` 平台下使用 `simd` 指令加速可以得到非常大的性能提升，无论是多核模式还是单核模式，`x86` 平台下会得到非常明显的性能提升，使用 `simd` 指令可以得到大概 `3~4` 倍的性能提升．但在 `arm64` 平台下，不太明白因为什么原因导致目前使用 `SIMD` 指令加速效果不是非常明显，后续还需要跟进，感觉很有可能是代码逻辑的问题,还是性能优化的问题, 无论是多核还是单核模式下，感觉使用 `simd` 指令始终感觉没有得到性能的大幅度提升，后期可以参考 `vectorscan` 的做法，将其代码中 `arm` 指令包装部分移植过来进行测试。

功能测试：

1. 测试用例的测试结果：

```
 Corpora generation failures:  0
umber of threads:  10 (5 scanner, 5 generator)
E2 scan alignment: 0
AILED: id 24013, libpcre compile failed: Failed to compile expression '(?:\x{dc\E)(?C)' (non-hex character in \x{} (closing brace missing?) at 8).

home/mike/workspace/hyperscan/hsbench-samples/test_cases/corpora/priority.txt
orpora processed:              245
scollider: The Pattern Collider Mark II
nly failed tests are displayed.

 Compilation failures:         pcre:0, ng:0, ue2:0
xpression path:    /home/mike/workspace/hyperscan/hsbench-samples/test_cases/pcre/passthrough.txt
orpora read from file: /home/mike/workspace/hyperscan/hsbench-samples/test_cases/corpora/options.txt
AILED: id 24013, corpora failure: no corpora found for pattern.
otal elapsed time: 0.616027 secs.
xpressions with failures:      0
uff.txt
```

尝试了一下 `hscollider` 的比对出错误，可能存在 `pcre` 版本的问题，部分正则语句编译提示错误，总体来说应该问题不太大，但是这个程序本身比较鸡肋，提示不够友好。所有的 `tools` 的工具代码也开发，实际可以根据自己的需要对这些工具进行二次开发。

问题现状

1. 华为鲲鹏提供的版本 chinema 部分程序在存在部分 BUG 测试用例无法通过，但是将该版本在 X86 平台下编译仍然存在问题，可能是该版本本身存的问题或者是某个库的版本不对，后期需要修正，目前的想法是在 hyperscan 5.4.0 的版本上进行修正。

- 目前在基于 hyperscan **5.2.0** 的版本已经可以完成替换 CPU SIMD 指令，但是目前内部还有 chinma 程序不管是在原生的 X86 平台和 ARM64 平台运行时都会有部分测试用例报错，但在最新的 **5.4.0** 版本就运行正常，这部分需要对比查找问题，到底是因为改编的 SIMD 部分引起的问题还是编译的变量设置引起的问题。

2. X86 平台编译效率太低：
   目前部分测试与编译工作需要在 X86 平台下运行，笔记本编译效率还是较低。

后期工作

- 尽快修正 chinema 的 bug ，升级到 5.4.0 版本上查看是否仍然存在该问题。
- 升级到 5.4.0 版本，需要再进行一次性能测试和功能性测试。
- 尝试编译运行 vectorscan 后期做替换，目前初步的设想是 hyperscan 官方提供的版本与社区版本 vectorsacn 都持续跟进。