Optimize Node.js for Noslate on Intel platforms

Author:	Intel, Alibaba
Version：	1.0.0
Date:	Feb 28, 2023

1 What is Noslate?

Noslate is an elegant, modern, and fully customizable serverless runtime developed by Alibaba. It contains Alibaba's node.js distribution, namely Anode which is targeting the default node.js for OpenAnolis. Node.js® is a JavaScript runtime-built framework on Chrome's V8 JavaScript engine.

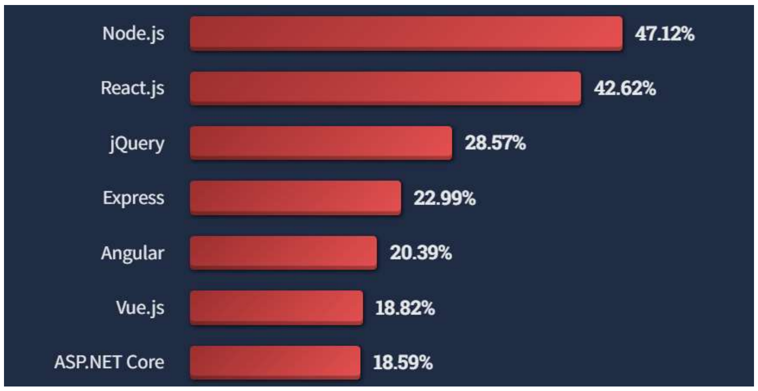
Noslate provides many additional advanced features, optimizations, and components for the cloud workloads, and it is opened sourced at https://github.com/noslate-project/noslate. Noslate contains following components:

- Anode: Node.js distribution for serverless https://github.com/noslate-project/node
- Aworker: Noslate JavaScript worker https://github.com/noslate-project/aworker
- Noslated: Workers management https://github.com/noslate-project/noslated
- Turf: Ultra-light container https://github.com/noslate-project/turf
- Debugger: V8 corefile debugger for gdb/lldb https://github.com/noslate-project/andb
- Arthur: Fast corefile generator https://github.com/noslate-project/arthur

This document provides the best practices and guideline for optimizing the performance and startup time of Noslate/node.js workload for Intel server platforms.

2 Node.js at Alibaba

According to stack overflow developer survey 2022, Node.js and React.js are the two most common web technologies used by Professional Developers and those learning to code.



Node.js has been largely used for server-side applications. The following areas are typical usages of Node.js:

1. Web/App server backend
2. Microservice and Function computing
3. Edge computing
4. IoT and Realtime communication

2.1 Noslate usage inside Alibaba

Node.js is one of the primary server application runtimes in Alibaba. Inside Alibaba, the employment of Node.js is twice larger than Golang's usage. Since 2018, the usage of Node.js in Alibaba has been increased by average 37% each year. Specifically, the number of active applications deployed to FaaS platforms (e.g., Aliyun Function Computing) has grown by average 140% per year.

More than three-thousand engineers in Alibaba are self-identified as a Node.js developer and actively programming in JavaScript. Well-known systems, like Taobao & Tmall, Amap, Alibaba Express, and Yuque, are the major users of Node.js server in the field. By the time of publishing, Node.js applications serve millions of requests per seconds inside Alibaba.

Noslate is a new pattern to deploy a scenario-guided optimized JavaScript application. It has hosted tens of thousands of customer-oriented server-side-rendering pages in the 11.11 Global Shopping Festival, 2022 and its usage is still growing inside Alibaba.

3 Optimization methods

Use typical workloads

To evaluate the optimizations, we need to use some representative workloads for the cloud real-life applications. In this document, several workloads have been employed for the performance analyzing and optimizing.

Score for full cores

As a server platform can have many CPU cores, the performance of Node.js workloads are measured in a way of fully utilizing all the cores. During measuring the performance, multiple Node.js docker instances are started.

Optimizations from multiple dimensions

For applications like this the performance of Node.js workloads can be Improved by many factors. There are a few optimization categories described in this document.

1. Pick up optimization patches already landed the latest node.js and turned on by default

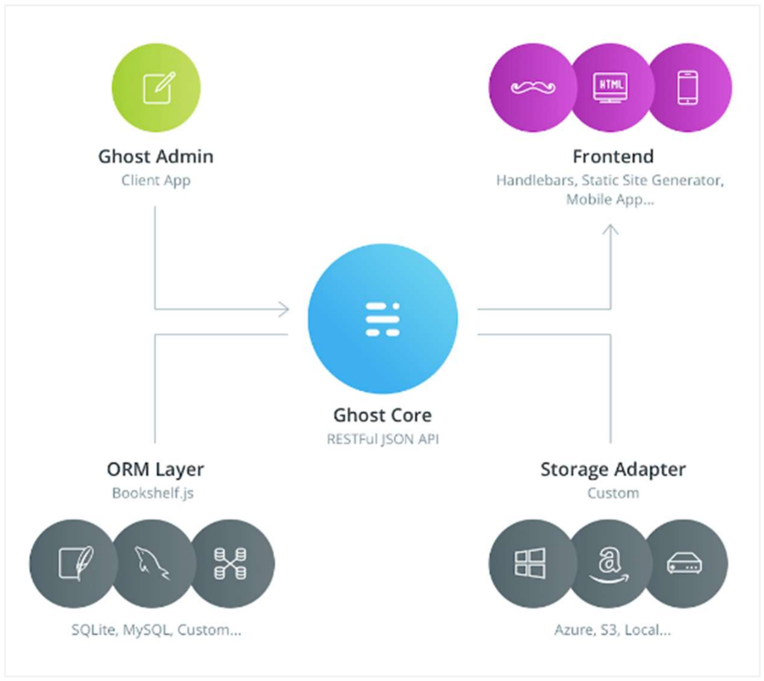
   Considering users may not use the latest Node.js, links to the patches are provided so anyone can check out the Node.js releases with the optimizations or pick them up for their current code base.
2. Building your Node.js binary with optimized configurations
3. Run Node.js with optimized command parameters
4. Optimize Docker configurations for the Node.js workloads
5. Optimize Nginx for Node.js workload use
6. Workload level optimizations for improving the startup time

4 Workloads

4.1 Ghost.js

Ghost.js (https://ghost.org/) is a popular open-source (MIT license) headless Node.js CMS designed for content publishing and blogging. It is structured as a modern, decoupled web application with a sensible service-based architecture. Ghost.js consists of several components:

- Core JSON API
- Admin client app
- Front-end theme layer



Ghost.js architecture diagram

The Ghost.js workload includes a Ghost.js application server (run by node.js), Nginx web server, MariaDB database server, and a ApacheBench client for stressing the test. And all those components are packed into single docker image. The throughput measured from client side is used for single instance performance.

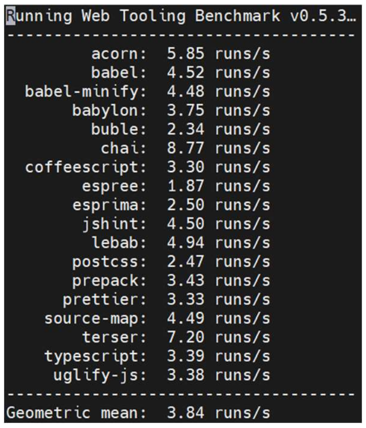
The test will launch many docker instances on the server platform (normally the same number of the server's logical core), and the sum of the throughput of all instances is the total performance. Note: this configuration is only for simplifying the test and it probably doesn't reflect the real situation.

The workload contains both http and https-based versions. For http workload, ApacheBench uses "http://" as protocol to connect Nginx, while "https://" is used for https workload. The https workload uses TLSv1.3, TLS_AES_256_GCM_SHA384 cypher suite and secp384r1 ssl_ecdh_curve algorithm.

4.2 WebTooling

WebTooling (https://v8.github.io/web-tooling-benchmark/) is a performance test suite focused on JavaScript execution with workloads from a variety of commonly used web developer tools.

It executes these core workloads a couple of times reports a single score at the end that balances them using geometric mean.



```
Running Web Tooling Benchmark v0.5.3…
--------------------------------------
         acorn:  5.85 runs/s
         babel:  4.52 runs/s
  babel-minify:  4.48 runs/s
       babylon:  3.75 runs/s
         buble:  2.34 runs/s
          chai:  8.77 runs/s
  coffeescript:  3.30 runs/s
        espree:  1.87 runs/s
       esprima:  2.50 runs/s
        jshint:  4.50 runs/s
         lebab:  4.94 runs/s
       postcss:  2.47 runs/s
       prepack:  3.43 runs/s
      prettier:  3.33 runs/s
    source-map:  4.49 runs/s
        terser:  7.20 runs/s
    typescript:  3.39 runs/s
     uglify-js:  3.38 runs/s
--------------------------------------
Geometric mean:  3.84 runs/s
```

Measurement method: Launch multiple docker instances (equal to logical core number in the server), in each docker instance, there is a Node.js process runs WebTooling benchmark, finally we will use the geometric mean of each docker instance multiplies instance count as performance results.

4.3 Function computing cold start up time

This workload continuously starts a FC framework on Node.js and ends it after the user function is executed. The performance score is defined as the average of start up time (latency).

This workload also uses the Docker to contain the workload. The test starts multiple instances of docker, and the final performance is calculated by the average latency across each container.

Please be noted this workload and opts are not for end user, but for CSPs to help to enhance their FC framework code to reduce start up latency.

4.4 Base64 Encoding micro bench

Base64 is a group of binary-to-text encoding schemes that represent binary data which is widely used in http, css and email. Node.js official code repo provides a benchmark containing a set of micro benches. Base64 encoding micro bench is part of the Node.js buffer benchmark (https://github.com/nodejs/node/blob/main/benchmark/buffers/buffer-base64-encode.js). It executes the Base64 encoding operation on a 64 bytes long buffer repeatedly and measures how many operations accomplished in one second (ops). According to the benchmark design it only runs single Node process for now.

5 Optimizations

5.1 AsyncNginx optimization

Optimization introduction

This optimization utilizes async mode Nginx which utilizes Intel® QuickAssist Technology (QAT) acceleration to improve the performance of https communication. QAT works under software mode, so QAT driver is not necessary to enable this optimization. It can increase the TPS of ghost.js https workload by +15% in ICX8358 platform in the workload test.

How to get it?

Follow the instructions in below link to enable this optimization step by step.

https://github.com/intel/asynch_mode_nginx

5.2 AVX512 for Base64 Encoding

Optimization introduction

This optimization utilizes AVX512VL and AVX512VBMI sub instruction sets (https://en.wikipedia.org/wiki/AVX-512) to optimize the Base64 encoding algorithms. In original algorithm, it reads in 4 input bytes and output 4 Base64 characters. With this optimization, it reads in 64 input bytes and output 64 Base64 characters with parallel bit operations.

How to get it?

- Apply patch in https://github.com/noslate-project/node/pull/9
- Build Noslate as usual

5.3 SIMD for Buffer swapping

Optimization introduction

This optimization utilizes SIMD, aka, SSE, AVX, AVX2, AVX512 VBMI to parallel the buffer swapping APIs Buffer.swap16(), Buffer.swap32() and Buffer.swap64(). This optimization works for buffer longer than 128 bytes to get the best optimization result.

How to get it?

- Apply patch in https://github.com/noslate-project/node/pull/10
- Build Noslate as usual

5.4 Huge page

Optimization introduction

This optimization utilizes Linux Transparent Huge-page (THP) feature to decrease the runtime TLB overhead.

How to get it?

- No patch file is needed
- run “configure --v8-enable-hugepage”, and then build the node binary

5.5 Semi-space size tuning

Optimization introduction

“--max_semi_space_size=128” is a runtime flag for V8, which sets the maximum semi-space size for V8's scavenge garbage collector as 128 MiB. Increasing the max size of a semi-space may bring throughput improvement for Node.js at the cost of more memory consumption (see #42511).

How to get it?

This is a runtime flag for Node.js, this is an example below:

```
$node --max_semi_space_size=128 app.js
```

5.6 Compress pointer

Optimization introduction

This optimization makes heap object in V8 smaller, which means it was used to reduce memory usage, but it also reduces cache and TLB pressure due to smaller object that also improves performance, but this option limits the max heap size to 4G.

How to get it?

The patches already landed Chromium/V8 and NodeJS upstream, but the optimization is not enabled for X64 platform by default.

We need to manually enable by the command below then build the binary.

```
$./configure --experimental-enable-pointer-compression
$make
```

5.7 Short built-in calls

Optimization introduction

This optimization makes jitted code closer to V8 code section for better branch prediction. The patch greatly reduces the long-range jumps to improve the IPC of workload execution.

How to get it?

- Apply the patch in https://github.com/noslate-project/node/pull/8
- Build Noslate as usual

5.8 Profile Guided Optimization (PGO)

Optimization introduction

This optimization utilizes workload's PGO profiling information to rebuild the optimized node binary.

How to get it?

The users need to finish the profiling with the workloads and build the node binary with the profiling data.

Follow the procedure:

- build the pgo-instrumented node binary:

```
$CC='gcc  -no-pie  -fno-PIE  -fno-reorder-blocks-and-partition  -fcf-protection=none -Wl,--emit-relocs -Wl,-znow '
$CXX='g++  -no-pie  -fno-PIE  -fno-reorder-blocks-and-partition  -fcf-protection=none -Wl,--emit-relocs -Wl,-znow '
$./configure --openssl-no-asm   --enable-pgo-generate
make
```

- Run the workloads with the above node binary to generate the profiling data files like in ~/pgodata. Rename it as ~/pgodata_wl_1. If multiple workloads run, they generate separate pgodata_wl_x.
- Merge the pgodata folder as all-in-one pgodata. Assume gcc-10 is installed:

```
$gcov-tool-10 merge ~/pgodata_wl_1/ ~/pgodata_wl_2 … -o ~/pgodata
```

- Build the pgo-optimized node binary with the profiling data files:

```
$CC='gcc  -no-pie  -fno-PIE  -fno-reorder-blocks-and-partition  -fcf-protection=none -Wl,--emit-relocs -Wl,-znow '
$CXX='g++  -no-pie  -fno-PIE  -fno-reorder-blocks-and-partition  -fcf-protection=none -Wl,--emit-relocs -Wl,-znow '
$./configure --enable-pgo-use
$make
```

5.9 Binary Optimization and Layout Tool (BOLT)

Optimization introduction

This optimization utilizes the BOLT tool to do binary optimization of the node.js binary.

How to get it?

- git clone https://github.com/facebookincubator/BOLT
- Follow the instructions in the repo to do optimization steps:
  - build bolt binaries,
  - use Linux-perf tool to collect the workload's profiling data,
  - then use bolt tool to generate the optimized node.js binary from the profiling data.

5.10 Container CPU constraints

Optimization introduction

This optimization adds "--cpuset-cpus=" option to docker run command line to apply cpu constraints for containers.

How to get it?

- No patch file is needed.
- Add --cpuset-cpus="$i,$i+1"option to docker run command line to make the running container use 2 vCPUs of number $i and $i+1. ($i is the container number: 0,1,2,..)

5.11 Node.js main thread CPU affinity

Optimization introduction

This optimization sets node.js main thread CPU affinity to current launching CPU core. Note: this opt is known helpful for reducing the TLB and Cache miss due to the thread migration in high CPU load situation. However, it may cause some side effects in other situations.

How to get it?

- Apply patch from https://github.com/noslate-project/node/pull/11
- Build Noslate as usual

5.12 Bytecode caching

Optimization introduction

This optimization demonstrates the Node.js workload utilizes V8 API to store the bytecode into disk at first run, and then reuse the bytecode in following runs which reduced part of parsing and compilation effort.

How to get it?

Function Computing framework source code is changed slightly. Follow the procedure:

1) Create a new JS file named "allcache.js"
2) allcache.js always checks the presence of bytecode cache of application. If it is

already present, load it without parsing and compiling the JS source. Otherwise, normal loading procedure is followed, then it should call V8 API of obj.createCachedData() to save the bytecode into the disk file.
3) allcache.js also overloads the Module.prototype._compile() function to intercept the JS loading procedure
4) Put the allcache.js file in the same folder of workload file "workload.js". In the first line of workload.js file, added below source line:

```
require("./allcache")
```

5) Run workload for one time to generate the bytecode cache.
6) From the second run, workload will consume the cache and start up time reduces.

6 Accumulative optimizations results

This section lists the overall performance improvement on Alibaba IceLake based cloud EC server after applying all available optimization for each workload.
In the tables below, "full server load" means the measurement is based on multiple Node instances running the same workload, there instance number is 64 (Hyper Thread number).

Alibaba cloud EC configurations

Platform	Alibaba IceLake based cloud EC server
CPU	Intel(R) Xeon(R) Platinum 8369B CPU @ 2.70GHz
Core number	32 core, 64 vCPU
Memory	256 GB
BIOS	SeaBIOS Version: 9e9f1cc
Microcode	microcode: 0x1, Stepping 6, model 106
OS	Ubuntu 20.04.5 LTS
Kernel	5.4.0-135-generic
Docker	23.0.1

Workload configurations

Software	Version
Noslate	V16.x
Ghost.js workload	V4.4.0
Nginx	V1.18.0
Async Nginx *	V0.4.5
QAT Engine *	V0.6.5
OpenSSL library *	V1.1.1j
IPP Crypto library *	2020u3
Ipsec Multi-Buffer library *	V0.55

SW with * only applied to Ghost https working mode.

Throughput benefit

noslate v16.x	Benefit
Webtooling (Full server load)	+52.0%
Ghost http (Full server load)	+53.7%
Ghost https (Full server load)	+49.7%
Base64 Encoding	+6%

FC start up time reduction

noslate v16.x	Reduction

FC start up time (Full server load)	-36.0%