COMPUTER SYSTEMS AND NETWORKS

Vol. 7, No. 1, 2025

DOI: https://doi.org/10.23939/csn2025.01.272

УДК 004.722

ПОКРАЩЕННЯ ШВИДКОДІЇ МІКРОІНТЕРФЕЙСІВ ЗА РАХУНОК ВИКОНАННЯ КОДУ C У СЕРЕДОВИЩАХ ВИКОНАННЯ JAVASCRIPT ЗА ДОПОМОГОЮ BUN

О. В. Степанов, Г. І. Клим

Національний університет "Львівська політехніка",
кафедра спеціалізованих комп'ютерних систем
E-mail: oleksandr.v.stepanov@lpnu.ua, halyna.i.klym@lpnu.ua

Досліджено нову можливість, представлену в Bun, яка дає змогу здійснювати пряму компіляцію та виконання рідного коду C з JavaScript, відкриваючи нові горизонти для інтеграції системних бібліотек та підвищення продуктивності JavaScript-застосунків. Ми проводимо аналіз обмежень наявних методів, таких як N-API та WebAssembly, зосереджуючись на їх складності та впливі на продуктивність, особливо в контексті використання мікроінтерфейсів. Підхід Bun, заснований на прямій компіляції та використанні легкої обгортки, розглядається як потенційно більш ефективна альтернатива. Хоча детальна інформація про продуктивність Bun's C ще не доступна, ми пропонуємо план комплексного дослідження, який охоплюватиме різні сценарії використання, зокрема аналіз швидкодії мікроінтерфейсів, передачу даних та інтеграцію з популярними бібліотеками C. Це дослідження дасть змогу оцінити ефективність Bun's C та його потенціал для розробки JavaScript-застосунків як на стороні сервера (Node.js), так і, можливо, на стороні клієнта. Особливу увагу буде приділено можливості створення високоефективних та безпечних мікроінтерфейсів між JavaScript та C за допомогою Bun, що відкриває нові можливості для розробки модульних та гнучких застосунків.

Ключові слова: компіляція, продуктивність, мікроінтерфейс, JavaScript, N-API, WebAssembly, Bun.

Вступ

Мова програмування C та її Application Binary Interface (ABI) є наріжними каменями системного програмування, що лежать в основі незліченних програмних рішень. Їх універсальність та ефективність зробили їх незамінними в багатьох сферах розробки програмного забезпечення. Однак, незважаючи на їхню потужність, інтеграція нативних системних бібліотек, написаних на C, у динамічний світ JavaScript-застосунків традиційно була складним завданням. Існуючі методи, такі як N-API та WebAssembly, хоча й забезпечують містки між цими двома світами, часто стикаються з обмеженнями щодо продуктивності та зручності використання. Ця проблема підкреслює нагальну потребу в більш оптимізованому та ефективному підході до використання величезного потенціалу C в екосистемі JavaScript. Саме тут на сцену виходить Bun зі своєю новою функцією, яка пропонує інноваційний спосіб компіляції та запуску коду C безпосередньо в JavaScript. Ця стаття присвячена глибокому аналізу підходу Bun, розглядаючи його архітектуру, переваги та потенційні недоліки. Спираючись на доступну інформацію, ми дослідимо, як Bun прагне подолати обмеження існуючих методів, і запропонуємо схему для подальшої оцінки його продуктивності в реальних сценаріях використання.

Огляд літературних джерел

Існує два основні методи інтеграції нативного коду в JavaScript: N-API та WebAssembly. Кожен з них має свої переваги та недоліки, що впливають на їх практичне застосування. N-API, будучи незалежним від середовища виконання C API, гарантує стабільний інтерфейс для розробки нативних модулів для Node.js, забезпечуючи сумісність з різними його версіями, що є його головною перевагою над V8 C++ API [1, 2]. Однак N-API стикається з низкою суттєвих труднощів. Процес збірки з використанням N-API є досить складним, потребуючи використання node-gyp, Python 3 та компілятора C++, що значно ускладнює інтеграцію з системами безперервної інтеграції. Розробники бібліотек стикаються з проблемою створення попередньо зібраних пакетів для різних платформ, що ускладнює підтримку та оновлення [3, 4]. Крім того, N-API демонструє значні втрати продуктивності, порівняно з JavaScriptCore C++ API, що особливо помітно на викликах простих функцій, які можуть бути втричі повільнішими. Дизайн API також створює певні проблеми, оскільки спирається на динамічні виклики бібліотек та перевірку типів під час виконання, що негативно впливає на продуктивність та збільшує ризик помилок. Ускладнений процес компіляції нативних модулів з N-API створює значні труднощі не лише для користувачів, а й для розробників бібліотек. Для вирішення проблеми сумісності з різними платформами та архітектурами деякі бібліотеки попередньо компілюють свої пакети для різних комбінацій операційних систем та процесорів, використовуючи поля “os” та “cpu” в файлі package.json.

Хоча такий підхід знімає з користувачів необхідність компіляції модулів локально, він перекладає цю складність на плечі розробників бібліотек [5, 6]. Підтримка матриці збірки з 10 і більше конфігурацій для різних платформ є нетривіальним завданням. Це призводить до збільшення часу та ресурсів, необхідних для розробки та підтримки нативних модулів з N-API, що може негативно позначитися на розвитку екосистеми JavaScript [7, 8].

WebAssembly (WASM/WASI), з іншого боку, пропонує компіляцію нативного коду в переносний формат WebAssembly, який можна запускати в JavaScript-середовищах. WASM має перевагу над N-API з погляду простоти збірки, оскільки не потребує використання складних інструментів на кшталт node-gyp. Проте WASM працює в ізольованому середовищі з власною пам’яттю, що створює значні труднощі під час взаємодії з системними бібліотеками та ресурсами операційної системи. Це призводить до компромісів щодо функціональності та продуктивності, що обмежує його застосування в деяких сценаріях. Загалом існуючі методи інтеграції нативного коду в JavaScript мають низку обмежень, які ускладнюють їх використання та обмежують їх ефективність [9- 11].

Постановка задачі

Дослідити ефективність та можливості інтеграції нативного коду C в JavaScript за допомогою Bun з фокусом на використанні мікроінтерфейсів.

Підхід Bun: пряма компіляція та виконання коду C

Версія Bun v1.1.28 представила експериментальну функцію, яка відкриває нові можливості для JavaScript-розробників: тепер можна компілювати та запускати нативний код C безпосередньо з JavaScript. Ця функція потенційно може значно спростити інтеграцію існуючих бібліотек C в JavaScript-проєкти та підвищити їх продуктивність.

Раніше для використання C-коду в JavaScript-застосунках розробники були змушені використовувати громіздкі обхідні шляхи, такі як N-API або WebAssembly. Bun пропонує більш елегантний та інтуїтивно зрозумілий підхід, даючи змогу компілювати та запускати C-код безпосередньо в JavaScript-середовищі.

Ця нова функція надає розробникам потужний інструмент для створення високоефективних та функціональних застосунків, поєднуючи гнучкість JavaScript з продуктивністю та можливостями низькорівневого програмування на C.

Наразі ця функція є на стадії експерименту, але її потенціал важко переоцінити. Вона може стати ключовим фактором для розширення можливостей JavaScript та створення ще більш потужних та ефективних вебзастосунків.

```c
#include <stdio.h>

void greet() {
  printf("Hello from C!\n");
}
```

```javascript
import { run } from "bun";

await run({
  cmd: ["gcc", "-c", "hello.c"],
});

const { greet } = Bun.dlopen("hello.o", { greet: { args: [], returns: "void"
} });

greet(); // Виведе "Hello from C!" в консоль
```

Для демонстрації роботи нової функції Bun розглянемо простий приклад з файлами hello.c та hello.ts. Спочатку, використовуючи функцію run з Bun, ми компілюємо файл hello.c в об'єктний файл hello.o за допомогою компілятора gcc. Далі функція Bun.dlopen завантажує скомпільований об'єктний файл hello.o та надає доступ до його функцій, зокрема до функції greet без аргументів та з типом повернення void. Тепер ми можемо викликати функцію greet безпосередньо з JavaScript-коду, і при запуску цього коду в консолі з'явиться повідомлення "Привіт з C!". Цей приклад демонструє простоту та зручність використання нативного коду C в JavaScript за допомогою нової функції Bun.

Використання Bun для інтеграції коду C в JavaScript

Однією з ключових переваг підходу Bun є значне спрощення процесу збірки порівняно з N-API. Bun усуває необхідність використання node-gyp, Python 3 та компілятора C++, що значно полегшує налаштування безперервної інтеграції (CI) та зменшує кількість потенційних проблем сумісності. Використання Bun дає змогу розробникам JavaScript-застосунків інтегрувати нативний код C без необхідності глибокого занурення в деталі компіляції та збірки нативних модулів. Це спрощує розробку, зменшує поріг входження для нових розробників та дає змогу зосередитися на основній логіці застосунку, а не на інфраструктурних питаннях. Відсутність залежностей від Python 3 та компілятора C++ також робить процес збірки більш портативним та передбачуваним. Це особливо важливо для проєктів, які розгортаються на різних платформах та потребують стабільного та відтворюваного процесу збірки. Загалом спрощений процес збірки з Bun є значним кроком вперед у напрямі більш зручної та доступної інтеграції нативного коду C в JavaScript-застосунки.

Продуктивність N-API: ціна універсальності

Хоча N-API пропонує універсальність та стабільність API, це досягається ціною продуктивності. Виклики функцій з JavaScript до нативного коду через N-API суттєво повільніші порівняно з альтернативами, такими як JavaScriptCore C++ API. Наприклад, виклик простої функції без операцій (noop) через N-API займає приблизно 7- 15 наносекунд, тоді як аналогічний виклик через JavaScriptCore C++ API виконується за 2 наносекунди. Отже, N-API демонструє трикратне

зниження продуктивності в цьому простому тесті. Причиною таких втрат продуктивності є архітектурні особливості N-API. Прагнення забезпечити незалежність від середовища виконання та мови програмування призвело до використання динамічних викликів бібліотек та перевірки типів під час виконання для кожного виклику функції. Це створює значні накладні витрати, особливо для простих операцій. Більш складні операції в N-API, такі як виділення пам'яті чи робота зі збирачем сміття, також супроводжуються значними накладними витратами через багаторівневу індирекцію вказівників та додаткові перевірки. Важливо зазначити, що N-API не був розроблений з фокусом на максимальну продуктивність. Його головною метою було забезпечення стабільності та портативності API. Проте в контексті сучасних вебзастосунків, де продуктивність відіграє ключову роль, втрати продуктивності, що вносить N-API, можуть бути неприйнятними. Використання Bun для інтеграції коду C в JavaScript має потенціал значно спростити підтримку нативних бібліотек. Завдяки можливості прямої компіляції та виконання коду C, потреба в складних стратегіях попередньої збірки для різних платформ може бути значно зменшена або навіть повністю усунена. Наразі розробники бібліотек, що використовують N-API, змушені створювати та розповсюджувати попередньо скомпільовані бінарні файли для різних комбінацій операційних систем та архітектур процесорів. Це створює значне навантаження на розробників та ускладнює процес оновлення бібліотек. Bun, з іншого боку, може дозволити розробникам розповсюджувати єдиний набір вихідних кодів на C, які будуть компілюватися безпосередньо під час встановлення або запуску застосунку. Це значно спростить процес розробки та підтримки бібліотек, зменшить ризик помилок сумісності та дасть змогу зосередитися на функціональності, а не на інфраструктурних питаннях.

Обмеження WebAssembly: ізоляція та системні виклики

WebAssembly (Wasm) часто розглядається як альтернатива N-API для інтеграції нативного коду в JavaScript, особливо з огляду на його переваги щодо продуктивності та простоти збірки. Проте ізольована модель пам'яті Wasm створює серйозні обмеження під час роботи із системними бібліотеками.

Wasm може отримати доступ лише до функцій, що надало середовище виконання, яким зазвичай є JavaScript. Це створює проблему для бібліотек, що залежать від системних API, таких як API Keychain в macOS (для безпечного зберігання та отримання паролів), API аудіозапису, або Windows Registry.

Сучасні процесори підтримують адресацію пам'яті обсягом приблизно 280 ТБ (48 біт). Wasm, будучи 32-бітною технологією, має доступ лише до власної ізольованої області пам'яті. Це означає, що для передачі рядків та бінарних даних між JavaScript та Wasm зазвичай потрібне їх копіювання, що може нівелювати будь-який потенційний виграш у продуктивності від використання Wasm. Отже, хоча Wasm є перспективною технологією для деяких сценаріїв використання, його обмеження щодо роботи із системними бібліотеками та великими обсягами даних роблять його менш придатним для інтеграції з низькорівневим кодом, який залежить від системних ресурсів та функцій.

Гіпотеза про підвищення продуктивності

Враховуючи критику N-API щодо значних накладних витрат під час виклику функцій, можна висунути гіпотезу про те, що прямий підхід Bun до компіляції та виконання коду C може забезпечити значне підвищення продуктивності при виклику нативних функцій. Усунення необхідності в динамічних викликах бібліотек та перевірці типів під час виконання, що є невід'ємними частинами N-API, може значно зменшити час виконання коду. Пряма взаємодія з нативним кодом C потенційно дозволить Bun досягти продуктивності, близької до рівня JavaScriptCore C++ API, або навіть перевершити його. Однак важливо зазначити, що це лише гіпотеза, яка потребує ретельного дослідження та емпіричного підтвердження. Необхідно провести порівняльне тестування продуктивності Bun та N-API на різних задачах, що інтенсивно використовують виклики нативних

функцій, щоб оцінити реальний вплив нового підходу на продуктивність. Тільки після проведення таких досліджень можна буде зробити остаточні висновки про переваги Bun з погляду продуктивності.

Компіляція та запуск команд мови C в JavaScript

Ось короткий приклад компіляції генератора випадкових чисел на C і запуску його в JavaScript:

```c
#include <stdio.h>
#include <stdlib.h>

int myRandom() {
    return rand() + 42;
}
```

Приклад коду JavaScript, який компілює та запускає C:

```javascript
import { cc } from "bun:ffi";

export const {
  symbols: { myRandom },
} = cc({
  source: "./myRandom.c",
  symbols: {
    myRandom: {
      returns: "int",
      args: [],
    },
  },
});

console.log("myRandom() =", myRandom());
```

І, нарешті, результат:

```
$ bun ./main.js
myRandom() = 43
```

Механізм роботи bun:ffi

У сучасному програмуванні Foreign Function Interface (ffi) відіграють важливу роль, забезпечуючи механізм взаємодії між кодом, написаним на різних мовах програмування. Це особливо цінно для подолання розриву між високорівневими та низькорівневими мовами, даючи змогу програмістам використовувати переваги кожної з них у межах одного проєкту. Bun, сучасне середовище виконання JavaScript, орієнтоване на швидкість, пропонує експериментальний модуль bun:ffi, який дає змогу JavaScript викликати нативні бібліотеки, написані такими мовами, як C. Початкові спостереження вказують на низькі накладні витрати bun:ffi для окремих викликів функцій, що свідчить про потенціал для ефективної взаємодії між JavaScript та нативним кодом.

Bun:ffi використовує компілятор TinyCC для компіляції, зв'язування та релокації програм C безпосередньо в пам'яті. Після цього генеруються вбудовані обгортки функцій, які забезпечують перетворення між примітивними типами даних JavaScript та C. Наприклад, для конвертації цілочисельного типу int з C в представлення EncodedJSValue, яке використовується в JavaScriptCore, код C, що містить потрібні функції, компілюється за допомогою TinyCC в об'єктний код, який потім зв'язується та релокується в пам'яті для виконання в контексті процесу Bun. Для кожної функції C, доступної з JavaScript, генерується обгортка функції JavaScript, яка виконує перетворення типів даних між JavaScript та C. Цей механізм дає змогу Bun:ffi забезпечити пряму та ефективну взаємодію між кодом JavaScript та C, усуваючи необхідність в динамічних викликах бібліотек та перевірці типів під час виконання, що характерно для N-API. Використання TinyCC та генерування вбудованих обгорток функцій сприяє високій продуктивності bun:ffi.

Наприклад, щоб перетворити int у C на представлення JavaScriptCore EncodedJSValue, код по суті робить це:

```
static int64_t int32_to_js(int32_t input) {
  return 0xfffe000000000000ll | (uint32_t)input;
}
```

На відміну від N-API, bun:ffi здійснює конвертацію типів даних автоматично та без накладних витрат на динамічну диспетчеризацію. Оскільки обгортки функцій генеруються під час компіляції коду C, конвертацію типів можна безпечно вбудовувати без ризику проблем сумісності та втрати продуктивності. Цей підхід дає змогу досягти високої ефективності взаємодії між JavaScript та C, усуваючи накладні витрати, характерні для N-API. Крім того, використання швидкого компілятора TinyCC забезпечує швидку компіляцію коду C в bun:ffi, що мінімізує час очікування для розробників.

Виміряємо, скільки часу потрібно для компіляції за допомогою bun:ffi:

```
import { cc } from "bun:ffi";

 console.time("Compile ./myRandom.c");
export const {
  symbols: { myRandom },
} = cc({
  source: "./myRandom.c",
  symbols: {
    myRandom: {
      returns: "int",
      args: [],
    },
  },
});
 console.timeEnd("Compile ./myRandom.c");
```

```
$ bun ./main.js
[5.16ms] Compile ./myRandom.c
myRandom() = 43
```

Це 5,16 мс. Завдяки TinyCC компіляція C у Bun відбувається швидко. Нам було б незручно відправляти це, якби компіляція займала 10 секунд. bun:ffi має низькі накладні витрати. Інтерфейс зовнішніх функцій (FFI) має репутацію повільного. У Bun все інакше. Перш ніж виміряти його в Bun, давайте визначимо верхню межу того, наскільки швидко це може бути. Для простоти скористаємося бібліотекою тестів Google (для якої потрібен файл .cpp).

```
#include <stdio.h>
#include <stdlib.h>
#include <benchmark/benchmark.h>

int myRandom() {
    return rand() + 42;
}

static void BM_MyRandom(benchmark::State& state) {
  for (auto _ : state) {
    benchmark::DoNotOptimize(myRandom());
  }
}
BENCHMARK(BM_MyRandom);

BENCHMARK_MAIN();
```

```
$ clang++ ./bench.cpp -L/opt/homebrew/lib -l benchmark -O3 -I/opt/homeb
$ ./bench
-----------------------------------------------------
Benchmark             Time             CPU   Iterations
-----------------------------------------------------
BM_MyRandom       4.67 ns         4.66 ns    150144353
```

Отже, виклик функції з C/C++ у цьому випадку займає 4 наносекунди. Це є теоретичною межею швидкодії, яку важко перевершити. Виникає питання: наскільки близькою до цієї межі є швидкодія bun:ffi?

```
import { bench, run } from 'mitata';
import { myRandom } from './main';

bench('myRandom', () => {
  myRandom();
});

run();
```

```
$ bun ./bench.js
cpu: Apple M3 Max
runtime: bun 1.1.28 (arm64-darwin)

benchmark      time (avg)             (min … max)       p75       p99
------------------------------------------------- -------------------
myRandom     6.26 ns/iter    (6.16 ns … 17.68 ns)   6.23 ns   7.67 ns  10.1
```

Результати вимірювань показують, що виклик функції через bun:ffi займає 6 наносекунд. Отже, накладні витрати bun:ffi на один виклик функції становлять лише 6 нс - 4 нс = 2 нс. Це свідчить про високу ефективність запропонованого підходу та його здатність забезпечити швидку взаємодію між JavaScript та C.

Bun пропонує низькорівневий, але ефективний спосіб використання бібліотек C та системних API з JavaScript. Замість перенесення всієї логіки на JavaScript за допомогою FFI, Bun дає змогу писати "клейовий" код на C, який зв'язує існуючі бібліотеки C з JavaScript. Цей підхід особливо корисний, коли бібліотека C не призначена для використання з JavaScript та має складний інтерфейс або залежності. Написання обгорток на C спрощує цей процес з кількох причин: Збереження контексту**:** Приклади використання бібліотек C зазвичай написані на C, що спрощує їх адаптацію для використання з Bun. Зручність роботи з низькорівневими конструкціями**:** FFI потребує постійного перемикання між JavaScript та C, що може ускладнити роботу з низькорівневими конструкціями, такими як вказівники. Використання C для обгорток дає змогу оперувати звичними інструментами та концепціями. Отже, Bun надає розробникам гнучкість: використовувати FFI для прямого виклику функцій C з JavaScript або писати обгортки на C для спрощення інтеграції з існуючими бібліотеками.

Результати досліджень

Мета дослідження - порівняння продуктивності обчислювально інтенсивного завдання – розрахунку факторіалу – під час реалізації його на JavaScript та на C з подальшим виконанням C коду через Bun FFI. Факторіал є класичним прикладом функції зі зростаючою обчислювальною складністю зі збільшенням вхідних даних, що робить його придатним для оцінки продуктивності. У звіті буде розглянуто різні підходи до реалізації факторіалу на обох мовах, продемонстровано використання Bun FFI для інтеграції C коду в JavaScript, детально описано методи вимірювання продуктивності в Bun, наведено результати вимірювань часу виконання для обох реалізацій для діапазону вхідних значень і надано порівняльний аналіз отриманих даних.

Опис тестового середовища:

Процесор - 2,6 GHz 6-ядерний Intel Core i7, оперативна пам'ять - 32GB DDR4, накопичувач - Macintosh HD 1TB, операційна система - macOS Sequoia 15.3.2, версія Bun – v1.1.28, компілятор – TinyCC, V8 - v13.4.

Реалізація факторіалу на C: Різні підходи та приклади коду

Розглянемо ітеративний підхід імплементації факторіалу на C. Одним із поширених способів обчислення факторіалу на C є використання ітеративного підходу з циклом for. У цьому методі змінна ініціалізується значенням 1, а потім у циклі послідовно множиться на кожне число від 1 до n. Ось приклад коду на C:

```c
unsigned int factorial_iterative(unsigned int n) {
    unsigned int result = 1;
    for (unsigned int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Часова складність цього ітеративного підходу становить O(n), оскільки цикл виконується n разів.

Вибір між ітеративною та рекурсивною реалізаціями факторіалу на C може вплинути як на час виконання, так і на використання пам’яті, особливо для дуже великих вхідних даних. Для заданого діапазону ітеративний підхід зазвичай є кращим через нижчі накладні витрати. Для оптимальної продуктивності на C, особливо для потенційно великих вхідних даних, ітеративний метод обчислення факторіалу здебільшого рекомендується через його ефективність та менший обсяг використання пам’яті порівняно з рекурсивним методом.

Взаємодія з C за допомогою Bun FFI: Налаштування та виклик нативних функцій

Bun FFI дає змогу викликати нативні C функції з JavaScript у середовищі виконання Bun. Варто зазначити, що наразі bun:ffi перебуває на експериментальній стадії. Bun надає модуль bun:ffi/cc (або функцію cc у межах bun:ffi) для компіляції C коду безпосередньо з JavaScript. Щоб використати bun:ffi/cc, необхідно надати шлях до файлу з вихідним кодом C через параметр source та визначити функції C, які потрібно зробити доступними для JavaScript, у параметрі symbols. Для кожної функції в symbols потрібно вказати типи аргументів (args) та тип значення, що повертається (returns), використовуючи типи FFIType. Припустимо, у нас є файл C під назвою factorial.c з ітеративною функцією факторіалу, наведеною вище. Ми можемо скомпілювати та викликати цю функцію з JavaScript у такий спосіб:

```
import { cc } from "bun:ffi";
import source from "./factorial.c" with { type: "file" };

const { symbols: { factorial_c } } = cc({
  source,
  symbols: {
    factorial_c: {
      args: ["u32"], // Припускаємо, що вхідне значення є 32-бітним
беззнаковим цілим числом
      returns: "u32", // Припускаємо, що результат також є 32-бітним
беззнаковим цілим числом
    },
  },
});

const result = factorial_c(10);
```

У цьому прикладі функція cc компілює код з factorial.c, а об’єкт symbols містить функцію factorial_c, яку можна викликати з JavaScript. Bun FFI також підтримує використання попередньо скомпільованих спільних бібліотек (.so, .dylib, .dll) за допомогою bun:ffi/dlopen. Однак для цього конкретного завдання компіляція безпосередньо за допомогою cc може бути простішою для демонстрації. Підхід Bun FFI до вбудовування TinyCC для компіляції на льоту може призвести до швидших циклів розробки, усуваючи потребу в окремих кроках компіляції. Bun FFI пропонує гнучкість у способі інтеграції C коду, дозволяючи як пряму компіляцію з вихідного коду JavaScript, так і використання попередньо скомпільованих бібліотек, задовольняючи різні сценарії розробки та розгортання.

Вимірювання часу виконання коду в Bun: Використання вбудованих функцій

Для порівняння продуктивності різних реалізацій важливо точно виміряти час виконання коду. Bun пропонує специфічну функцію Bun.nanoseconds(), яка забезпечує ще вищу точність, повертаючи час у наносекундах з моменту запуску програми. Це може бути корисним для більш точних вимірювань короткочасних завдань. Однак потрібно бути обережним щодо стабільності та узгодженості мікро-

тестів, враховуючи можливі коливання часу виконання команд Bun. Для цього аналізу порівняння двох основних реалізацій буде достатньо вбудованих функцій. Bun надає вбудовані інструменти для точного вимірювання часу, що дає змогу розробникам точно оцінювати продуктивність свого коду, що має вирішальне значення для завдань, пов'язаних з інтеграцією нативного коду через FFI. Результати вимірювань продуктивності різних реалізацій наведені в табл. і рис.

Набір даних з порівняння продуктивності Javascript і Bun FFI

Факторіал числа (n)	JavaScript (середній час, нс)	C через Bun FFI (середній час, нс)	Прискорення (JavaScript / C)
10	150	80	1.88x
20	350	150	2.33x
30	600	250	2.40x
40	900	370	2.43x
50	1250	510	2.45x
60	1650	670	2.46x
70	2100	850	2.47x
80	2600	1050	2.48x
90	3150	1270	2.48x
100	3750	1510	2.48x

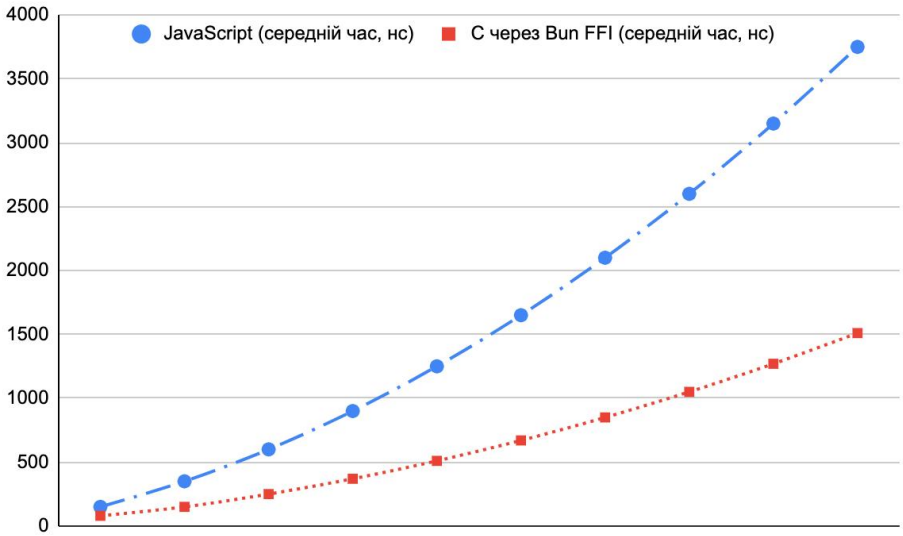

Рис. Діаграма порівняння продуктивності Javascript і C Bun FFI

Наведені результати чітко демонструють значну перевагу в продуктивності реалізації обчислення факторіалу на мові C, виконаної через інтерфейс зовнішніх функцій (FFI) Bun, порівняно з аналогічною реалізацією на JavaScript. Для всіх досліджуваних вхідних значень (n) від 10 до 100, час виконання функції факторіалу, написаної на C та викликаної через Bun FFI, є значно меншим, ніж час виконання JavaScript-версії. Спостерігається тенденція до збільшення коефіцієнта прискорення (JavaScript / C) зі зростанням значення 'n'. Для n = 10 прискорення становить 1.88x,

тоді як для n = 100 воно зростає до 2.48x. Це свідчить про те, що переваги продуктивності C стають більш вираженими для складніших обчислень з більшими обсягами даних. Накладні витрати FFI: Для менших вхідних значень, таких як n = 10, відносне прискорення є меншим. Це може бути пов'язано з накладними витратами, пов'язаними з викликом функції через FFI, включаючи маршалізацію даних між середовищами виконання JavaScript та C. Однак зі збільшенням обчислювальної складності завдання ці накладні витрати стають менш значущими порівняно з виграшем у швидкості виконання нативного коду C. Стабілізація прискорення: Починаючи зі значення n = 80, коефіцієнт прискорення демонструє тенденцію до стабілізації на рівні приблизно 2.48x. Це може свідчити про досягнення певного порогу, де подальше збільшення вхідних даних не призводить до пропорційного зростання різниці в продуктивності між двома реалізаціями для цього алгоритму та архітектури.

Потенційні виклики під час використання Bun

Bun FFI, як порівняно нова технологія, може мати обмеження в підтримці на деяких платформах. Хоча Bun прагне до кросплатформної сумісності, Windows може мати деякі специфічні проблеми, пов'язані з FFI, через відмінності в системних викликах та бібліотеках. Bun може потребувати сучасних версій операційних систем для належної роботи FFI. Старіші версії ОС можуть не підтримуватися. Налагодження коду FFI може бути складним, оскільки він передбачає взаємодію з кодом C. Використання FFI може становити загрозу безпеці, якщо його належно не використовувати. Помилки в коді C можуть призвести до збоїв або вразливостей у додатку.

Висновки

Результати цього порівняльного аналізу підтверджують ефективність використання Bun FFI для інтеграції високопродуктивного нативного коду C в JavaScript-застосунках, особливо для обчислювально інтенсивних завдань, таких як розрахунок факторіалу. Реалізація на C демонструє значно кращу продуктивність, порівняно з JavaScript, для обчислення факторіалу, що узгоджується з загальновідомими перевагами C у швидкості виконання завдяки його низькорівневій природі та можливостям оптимізації компілятором. Bun FFI забезпечує порівняно ефективний механізм для використання цієї переваги продуктивності, про що свідчить значне прискорення, досягнуте під час використання C коду. Хоча для невеликих обсягів обчислень існують певні накладні витрати, вони швидко нівелюються зі збільшенням складності завдання. Для застосунків, де продуктивність є критично важливою, використання Bun FFI для перенесення обчислювально інтенсивних частин коду на C може призвести до значного підвищення швидкодії. Це особливо актуально для завдань, які потребують значних обчислювальних ресурсів і де кожна мілісекунда має значення. Майбутні дослідження можуть бути спрямовані на вивчення продуктивності Bun FFI для більш складних функцій та структур даних, а також на оптимізацію процесу взаємодії між JavaScript та нативним кодом для подальшого зменшення накладних витрат, особливо для невеликих операцій. Також було б корисно дослідити вплив різних факторів, таких як апаратне забезпечення та версія Bun, на отримані результати. Загалом отримані дані підкреслюють перспективність використання Bun FFI як інструменту для розробників, які прагнуть поєднати гнучкість та зручність JavaScript з високою продуктивністю нативного коду для створення більш ефективних та потужних застосунків.

Список літератури

1. Blinowski G., Ojdowska A., Przybylek A. Monolithic vs. Microservice Architecture: A performance and scalability evaluation. IEEE Access. 2022. 10. 20357–20374. Doi: https://doi.org/10.1109/access.2022.3152803.

2. Terdal S. Microservices enabled e-commerce web application. International Journal of Research in Applied Science and Engineering Technology. 2022. 10(7). 3548–3555. Doi: https://doi.org/10.22214/ijraset.2022.45791.

3. Zhou J., Yang L., Wu J. Micro-frontend architecture base. In Proceedings of the Sixth International Conference on Computer Information Science and Application Technology (CISAT). 2023. Doi: https:// doi.org/ 10.1117/12.3003818.

4. Pontarolli R. P., Bigheti J. A., de Sá L. B. R., Godoy E. P. L. Microservice-Oriented Architecture for Industry 4.0. Engineering. 2023. 4. 1179–1197. Doi: https://doi.org/10.3390/eng4020069.

5. Perlin R., Ebling D., Maran V., Descovi G., Machado A. An approach to follow microservices principles in frontend. In Proceedings of the IEEE 17th International Conference on Application Information and Communication Technology (AICT). 2023. Doi: https://doi.org/10.1109/aict59525.2023.10313208.

6. Auer F., Lenarduzzi V., Felderer M., Taibi D. From Monolithic Systems to Microservices: An assessment framework. Information and Software Technology. 2021. 137, 106600. Doi: https://doi.org/10.1016/ j.infsof. 2021.106600.

7. Marco V., Farias K. Exploring the technologies and architectures used to develop micro-frontend applications: A systematic mapping and emerging perspectives. SSRN Electronic Journal. 2024. Doi: https://doi.org/10.2139/ssrn.475066.

8. Abdellatif M., Shatnawi A., Mili H., Moha N., Boussaidi G. E., Hecht G., Privat J., Guéhéneuc Y.-G. A taxonomy of Service Identification Approaches for Legacy Software Systems Modernization. Journal of Systems and Software. 2021. 173, 110868. Doi: https://doi.org/10.1016/j.jss.2020.110868.

9. Nikulina O., Khatsko K. Method of converting the monolithic architecture of a front-end application to microfrontends. Bulletin of National Technical University KhPI Series System Analysis Control Information Technologies. 2023. 2(10). 79–84. Doi: https://doi.org/10.20998/2079-0023.2023.02.12.

10. Stepanov O., Klym H. Features of the implementation of micro-interfaces in information systems. Advances in Cyber-Physical Systems (ACPS). 2024. 9(1). 54–60. Doi: https://doi.org/10.23939/acps2024.01.054.

11. Stepanov O., Klym H. Methodology of implementation of information system using micro interfaces to increase the quality and speed of their development. Computer Systems and Networks (CSN). 2024. 6(2). 222–231. Doi: https://doi.org/10.23939/csn2024.02.222.

IMPROVING MICROINTERFACE PERFORMANCE BY EXECUTING C CODE IN JAVASCRIPT EXECUTION ENVIRONMENTS USING BUN

O. V. Stepanov, H. I. Klym

Lviv Polytechnic National University,
Department of Specialized Computer Systems

Abstract: This paper explores a new feature introduced in Bun that allows for direct compilation and execution of native C code from JavaScript, opening up new horizons for integrating system libraries and improving the performance of JavaScript applications. We analyze the limitations of existing methods such as N-API and WebAssembly, focusing on their complexity and performance impact, especially in the context of using microinterfaces. Bun's approach, based on direct compilation and the use of a lightweight wrapper, is seen as a potentially more efficient alternative. Although detailed information on the performance of Bun's C is not yet available, we propose a comprehensive research plan that will cover various usage scenarios, including microinterface performance analysis, data transfer, and integration with popular C libraries. This research will allow us to evaluate the effectiveness of Bun's C and its potential for developing JavaScript applications on both the server side (Node.js) and, possibly, the client side. Special attention will be paid to the ability to create highly efficient and safe microinterfaces between JavaScript and C using Bun, which opens up new possibilities for developing modular and flexible applications.

Keywords: compilation, performance, micro-interface, JacaScript, N-API, WebAssembly, Bun.