// Rust in der Cloud: Performance-Vergleich mit TypeScript und Java in AWS-Lambda

Rust | Cloud | AWS | Serverless | Node.js | Java | JavaScript

Nicolas Großmann

20.6.2024 | 6 Minuten Lesezeit

In diesem Artikel setzen wir Rust ein, um AWS-Lambda-Funktionen zu implementieren und vergleichen die Performance mit TypeScript (Node.js) und Java (JVM).

Rust ist momentan in aller Munde und wird für seine Performance, Effizienz und Speichersicherheit gefeiert. Unser Kollege Goetz hat in früheren Artikeln Rust bereits im Hinblick auf die Nachhaltigkeit und den Einsatz in der Full Stack Web Entwicklung beschrieben. Spannend wird es daher, nun auch zu testen, wie sich Rust in der Cloud schlägt. Als konkretes Beispiel werden wir uns dafür AWS-Lambda ansehen.

hiedene Runtimes für AWS-Lambda

Die Serverless Funktionen von AWS können mit verschiedenen Runtimes ausgeführt werden. Häufig werden hier Sprachen eingesetzt, die den Einsatz einer virtuellen Maschine erfordern. TypeScript (bzw. JavaScript) ist eine beliebte Wahl, weil diese Sprache relativ leicht zu erlernen ist und von vielen beherrscht wird. Jedoch ist die Ausführung über Node.js relativ langsam, da der Code zur Laufzeit Zeile für Zeile interpretiert wird. Auch Java hat durch den Einsatz der JVM einen Overhead, der nicht zu vernachlässigen ist.

Dagegen wird Rust zu Maschinencode kompiliert und kann somit direkt ausgeführt werden. Zudem verwendet Rust keinen Garbage Collector. Es ist also spannend herauszufinden, welches Optimierungspotenzial Rust in diesem Umfeld bietet.

Bereitstellung und cargo lambda

Ein großer Vorteil bei dem Einsatz von TypeScript und dem AWS CDK ist, dass sämtlicher Code in TypeScript geschrieben werden kann und keine weitere Kompilierung vor dem Deploy notwendig ist. Demnach müsste mit Rust zunächst ein Kompilat erstellt werden, welches dann als Zip-Datei der Lambda zugewiesen wird. Glücklicherweise hat AWS mit cargo lambda ein Tool bereitgestellt, das das Deployment vereinfacht und die zusätzlichen Schritte unter einem einzelnen Command versteckt. Damit ist der zusätzliche Aufwand aus Sicht der Entwickler*innen überschaubar.

Speichergrößen

Für eine AWS-Lambda-Funktion lässt sich die zugewiesene Speichergröße einstellen. Je mehr Speicher wir zuweisen, desto teurer wird die Nutzung. Das Preismodell sieht vor, dass sowohl die Ausführungsdauer als auch die zugewiesene Speichergröße den Preis einer Ausführung bestimmen.

Ein wichtiger Zusammenhang darf jedoch nicht vergessen werden: AWS erhöht m[illegible]r Speicher auch die zugewiesenen virtuellen CPU-Ressourcen (siehe D[illegible]ntation). Demnach kann es vorkommen, dass Funktionen mit mehr

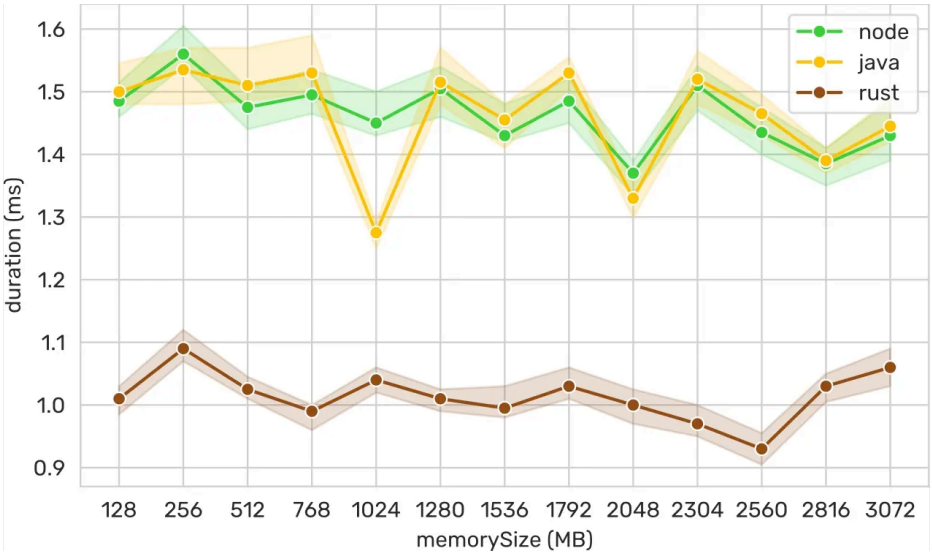
zugewiesenem Speicher günstiger ausfallen, wenn die zusätzliche CPU-Leistung die Ausführungsdauer erheblich reduziert.

Im Zusammenhang mit der speichereffizienten Arbeitsweise von Rust ist es daher auch sehr interessant, die entstehenden Kosten zu analysieren.

Benchmarking

Um die Performance der Sprachen miteinander zu vergleichen, haben wir verschiedene Funktionen in den jeweiligen Sprachen implementiert, mehrfach ausgeführt und dann in Form von Diagrammen ausgewertet. Um korrekte Daten zu erhalten, wurden die von AWS bereitgestellten CloudWatch-Logs herangezogen und die Metriken `@duration` und `@maxMemoryUsed` ausgewertet.

Zunächst wurde eine klassische **Hello-World-Funktion** umgesetzt, die keine große Arbeit verrichtet, aber dennoch als Baseline dient.

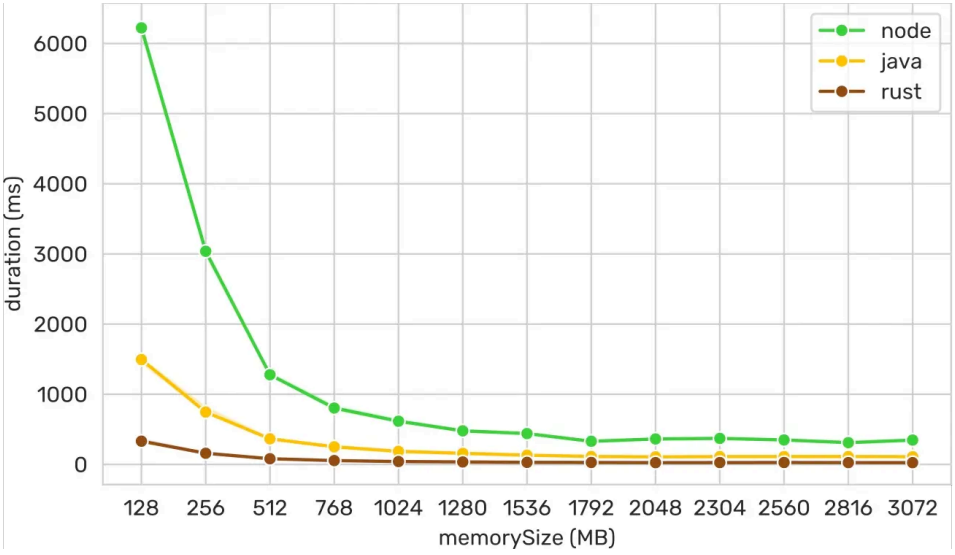


Auch wenn wir hier von Bruchteilen von Millisekunden sprechen, ist es dennoch interessant zu sehen, dass sich die Ausführungsdauer von Rust deutlich von der von Node.js und Java abhebt.

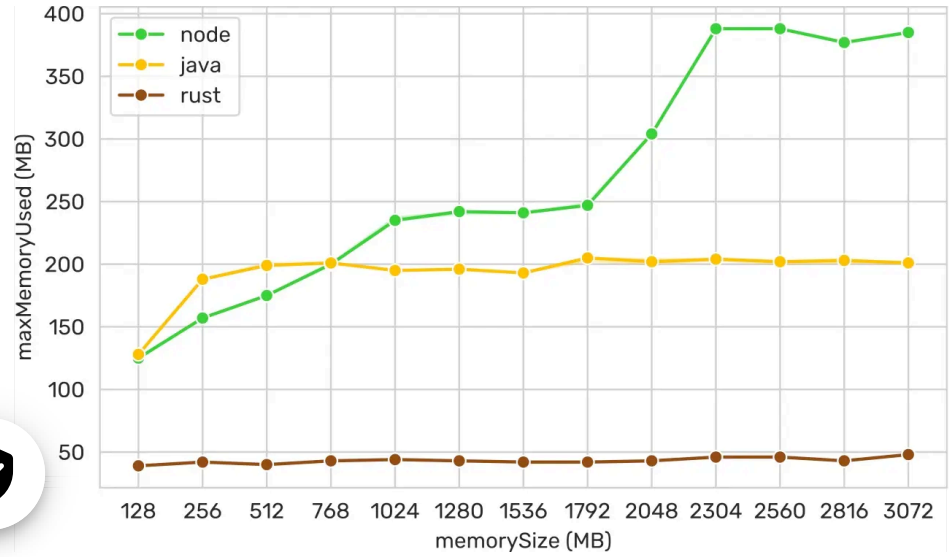
Die Wahrheit ist aber auch, dass Hello-World-Funktionen nicht wirklich repräsentativ sind. Daher wird im Folgenden ein weiterer Benchmark

durchgeführt, der auch die Kommunikation mit einem weiteren AWS-Service beinhaltet.

Die Lambda-Funktion fragt eine JSON-Datei aus einem S3-Bucket ab, die ein Array von Zahlenpaaren enthält. Auf Grundlage dieser wird innerhalb der Lambda-Funktion der rechenintensive k-Means-Algorithmus angewandt, der als Rückgabewert die ermittelten Clusterzentren zurückgibt.

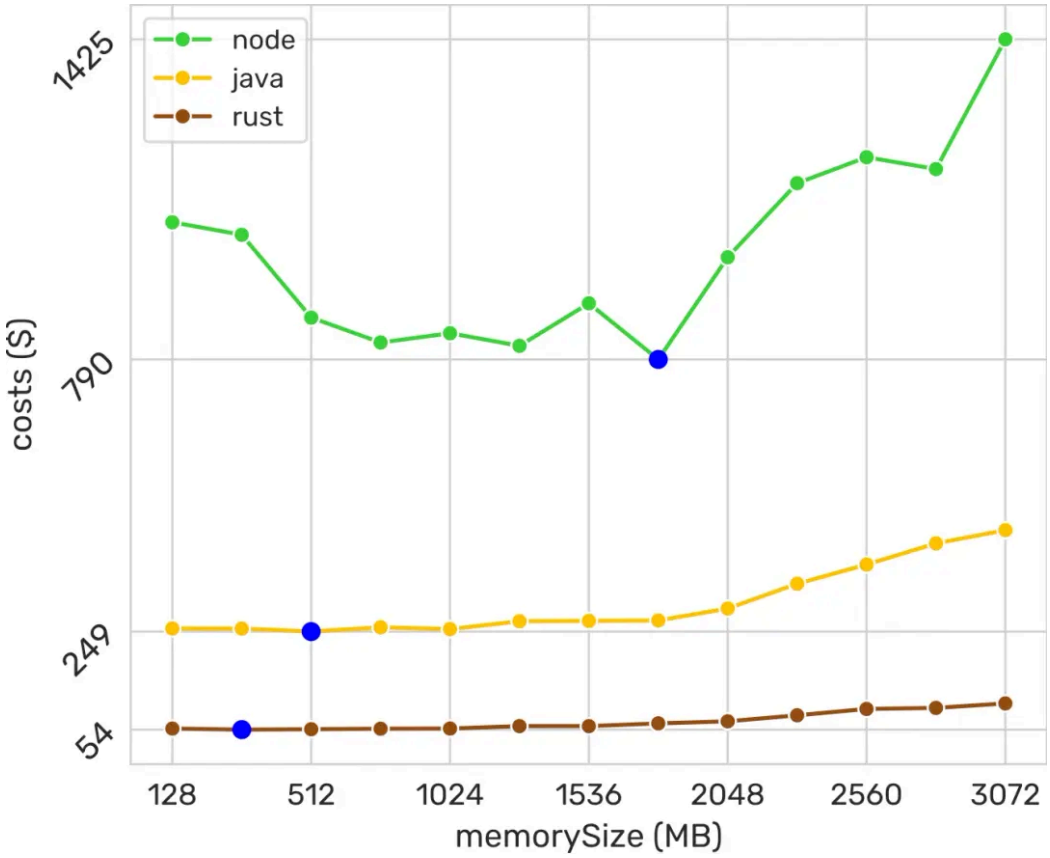


Hier wird der Zusammenhang von Speichergröße und CPU-Leistung deutlich, wovon alle Runtimes profitieren. Auch bei dieser Funktion hebt sich Rust deutlich von den anderen ab und führt die Funktion schneller aus. Neben der Ausführungsdauer ist auch der verbrauchte Speicher interessant.



Rust bleibt konstant und unbeeindruckt von mehr verfügbarem Speicher, während Node.js und Java tendenziell mehr Speicher verwenden, wenn mehr Speicher zur Verfügung steht (insbesondere bei kleinen Speichergrößen). Insgesamt zeigt sich, dass Rust sehr viel speichereffizienter arbeitet.

Doch wie verhalten sich nun die Kosten? Dazu werden im nachfolgenden Diagramm die Kosten gegenübergestellt, wobei von 100.000.000 Ausführungen ausgegangen wird.

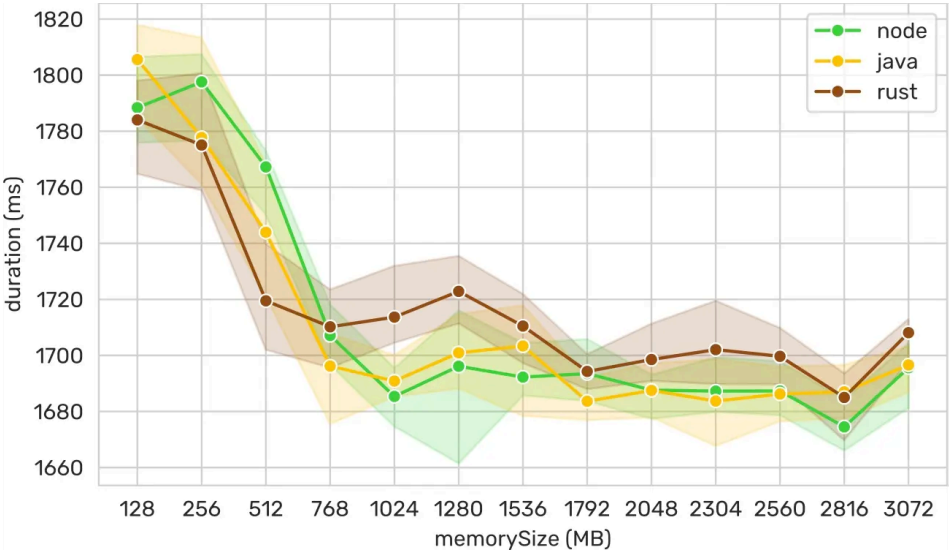


Es zeigt sich, dass Rust bereits mit weniger zugewiesenem Speicher die minimalen Kosten erreicht, was die effiziente Speichernutzung von Rust hervorhebt. Dahingegen erreicht Node.js das Kostenminimum erst mit mehr verfügbaren Speicher. Analog zur Ausführungsdauer sind die Kosten mit Rust deutlich reduziert.

Grenzen des Optimierungspotenzials

Trotz der positiven Ergebnisse gibt es für Rust auch Grenzen. Eine Umstellung

auf Rust kann nur den Anteil einer Lambda-Funktion beschleunigen, der tatsächlich von der Programmiersprache beeinflusst wird. Wird die meiste Zeit für eine Datenbankabfrage benötigt, kann dieser Teil auch von Rust nicht beschleunigt werden. Dies wird deutlich bei der Implementierung einer Funktion, die im IoT-Umfeld zum Einsatz kommt. Diese fragt Daten aus einer Timestream-Datenbank ab, verpackt sie in ein kompaktes Array und gibt sie zurück.



Hier zeigt sich, dass Rust weder signifikant schneller noch langsamer als Node.js oder Java ist. Alle drei Implementierungen sind ungefähr gleich schnell. Dies liegt daran, dass die Datenbankabfrage selbst den mit Abstand größten (zeitlichen) Anteil an der Funktion innehat. Die Auswahl der Programmiersprache für die Lambda-Funktion hat auf diesen Teil keinen Einfluss, weswegen die Ergebnisse wenig überraschend sind. Es verdeutlicht aber, dass Rust nicht in allen Anwendungsfällen und ausnahmslos Optimierungspotenzial bietet. Das Optimierungspotenzial hängt immer vom genauen Funktionsablauf und den verwendeten Bibliotheken ab.

Cold-Start-Problematik

Ein bekanntes Problem im FaaS-Umfeld ist die sogenannte Cold-Start-Problematik. Bei der Ausführung von AWS-Lambda-Funktionen muss zunächst der zugehörige Code heruntergeladen und die Ausführungsumgebung gestartet werden. Erst dann wird der eigentliche

Handler-Code ausgeführt. Diese Initialisierungsdauer fällt immer dann an, wenn keine warme Ausführungsumgebung zur Verfügung steht. Auch diese Initialisierungsdauer soll zwischen den drei Runtimes untersucht werden.

Dazu haben wir die genannten Funktionen erneut und als Cold-Start ausgeführt. Nachfolgend haben wir die Metrik `@initDuration` ausgewertet, die von CloudWatch bereitgestellt wird und nur jene Ausführungsdauer (in ms) beinhaltet, die zum Initialisieren benötigt wurde. Die tatsächliche Ausführungsdauer des Handler-Codes ist darin nicht enthalten.

Funktion	Runtime	128 MB	768 MB	1536 MB	2304 MB	3072 MB
Minimalfunktion	node	138	138	140	140	139
	java	430	433	427	353	335
	rust	15	14	15	14	14
k-Means	node	386	388	381	363	372
	java	1443	1421	1420	1239	1189
	rust	59	56	56	57	58
Timestream-Abfrage	node	321	317	324	300	299
	java	1674	1675	1642	1377	1347
	rust	55	51	51	52	52

Es ist klar zu erkennen, dass die Initialisierungsdauern von Rust deutlich kürzer sind als jene von Node.js und Java. Insbesondere bei Java benötigt für die Initialisierung relativ lange, da hier die JVM gestartet werden muss.

Somit bietet Rust ein enormes Optimierungspotenzial hinsichtlich der Cold-Start-Problematik. Sind unsere Anwendungen zeitkritisch bzw. würden unter einem langen Cold-Start leiden, so kann sich der Einsatz von Rust lohnen.

Fazit

Wir konnten sehen, dass der Einsatz von Rust in AWS-Lambda-Funktionen mit

erheblichen Vorteilen einhergeht. Die schnellere Ausführungsdauer und der effizientere Speicherverbrauch sorgen auch hinsichtlich der Kosten für eine Verbesserung. Das Ausmaß dieser Verbesserung ist aber maßgeblich von der genauen Funktion abhängig und kann nicht ganz allgemein beziffert werden. Zudem sind mit Rust deutlich verkürzte Cold-Start-Initialisierungsdauern festgestellt worden, sodass wir mit dem Einsatz von Rust insbesondere diesbezüglich profitieren.

Insgesamt ist es spannend zu sehen, was durch eine Umstellung möglich ist und dass es sich lohnen kann, über solche Optimierungen nachzudenken.

Blog-Autor*in

Du hast noch Fragen zu diesem Thema? Dann sprich mich einfach an.

Kontakt aufnehmen

Nicolas Großmann

Praktikant

codecentric

// Dein Job bei codecentric?

Jobs

Agile Developer und Consultant (w/d/m)

Alle Standorte

Backend | Frontend | Fullstack

Zur Stellenanzeige

// Weitere Artikel in diesem Themenbereich

codecentric

Server Actions in Next.js 14

Server Actions wurden in Next.js 14 als neue Methode zum Senden von Daten an den Server eingeführt (siehe die Dokumentation). Es sind asynchrone Funktionen, die sowohl in Server-Komponenten, innerhalb von serverseitigen Forms, als auch in Client-Komponenten...

Webdevelopment React JavaScript

10.6.2024 | 8 Minuten Lesezeit

Lukas Lehmann

Willkommen in der nä

Während der jährlichen I
Atlassian die sogenannt
angekündigt, die ab dem
haben wir alle „verschied
eine war „Jira-Software'
Work Management...

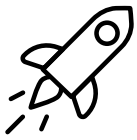
Cloud Atlassian

15.5.2024 | 4 Minuten Lese

Aurimas Brazaitis

// Gemeinsam bessere Projekte umsetzen.

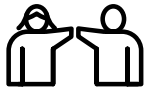
codecentric



Wir helfen deinem Unternehmen.

Du stehst vor einer großen IT-Herausforderung? Wir sorgen für eine maßgeschneiderte Unterstützung. Informiere dich jetzt.

Unsere Leistungen



Hilf uns, noch besser zu werden.

Wir sind immer auf der Suche nach neuen Talenten. Auch für dich ist die passende Stelle dabei.

Zu den Jobangeboten

codecentric


Management
System
ISO/IEC 27001:2013
TÜVRheinland
ZERTIFIZIERT
www.tuv.com
ID 9000031402


charta der vielfalt
UNTERZEICHNET


Allianz für
Cyber-Sicherheit
Teilnehmer


Zertifizierte
Corporation