Vergelijkende Prestatieanalyse van Java Virtual Machine (JVM) op Mainframes en Moderne PC's voor Computationeel Intensieve Algoritmen:

Een Onderzoek en vergelijkende studie.

Muhammed Baki Akin.

Scriptie voorgedragen tot het bekomen van de graad van
Professionele bachelor in de toegepaste informatica

Promotor: Dhr. Leendert Blondeel
Co-promotor: Dhr. Renault Gio
Academiejaar: 2023–2024
Eerste examenperiode

Departement IT en Digitale Innovatie .

HO
GENT

Woord vooraf

Als eerste zou ik graag een welgemeend dankwoord willen overbrengen aan mijn familieleden, die onvoorwaardelijk en ruimhartig hun steun hebben verleend, zowel emotioneel, mentaal als op verschillende andere manieren. Zij hebben mijn academische avontuur enorm vergemakkelijkt en mijn doorzettingsvermogen versterkt.
Ook zou ik graag ook mijn vrienden van harte willen danken voor hun waardevolle ondersteuning en advies tijdens het schrijfproces van dit onderzoek voor mijn bachelorproef. Hun aanmoedigingen en waardevolle suggesties maakten het mogelijk om de obstakels die ik tegenkwam te overwinnen en mijn werk te verbeteren.
Hierbij zou ik graag ook mijn co-promotor Renault Gio willen bedanken, wiens waardevolle kennis en steun gedurende het hele traject erg waardevol is geweest. Ik ben hem zeer dankbaar voor zijn bereidheid zijn kennis met mij te delen en zijn helpende hand aan te bieden wanneer dat nodig was. Ik wil ook mijn vriend en mede-collega van mijn copromotor, Mehmet Karademir, bedanken voor zijn voortdurende steun en medewerking.
Mijn laatste woord van dank gaat uit naar mijn promotor, Leendert Blondeel, voor zijn onschatbare hulp en advies tijdens dit proces.

U kunt hieronder mijn afstudeerscriptie vinden met de titel ”Vergelijkende prestatie analyse van de Java Virtual Machine (JVM) op Mainframes en moderne pc ’s voor reken intensieve algoritmen: Een onderzoek en vergelijkende studie”. Deze thesis is gemaakt in het kader van het behalen van mijn bachelor diploma Toegepaste Informatica, met als afstudeerrichting Mainframe Expertise.

Tijdens mijn specialisatie jaar mainframe heb ik vooral interactie en kennis opgedaan over mainframe computers en technologieën omtrent deze systemen. Door deze interactie met mainframe computers raakte ik meer geïnteresseerd in deze technologie. Vanwege deze interesse wilde ik een onderwerp kiezen dat gerelateerd was met mainframes. Na een aantal ideeën kwam ik tot de conclusie om het eerder genoemde onderwerp te kiezen en daar een bachelorproef rond te doen.
Gedurende het onderzoeksproces rond het gekozen onderwerp heb ik bepaalde zaken ontdekt die ik tijdens mijn werkgebaseerde opleiding als stage niet aan bod was gekomen. Dit zorgde ervoor dat ik mijn kennis over mainframetechnologieën kon uitbreiden buiten het curriculum dat ik op de campus kreeg. Deze bachelorproef heeft me in de gelegenheid gesteld om mijn kennis over het mainframe uit

te breiden en heeft me ook waardevolle informatie opgeleverd.
Tot besluit wens ik u, als lezer, hierbij veel leesplezier en hoop ik dat u door de informatie beschreven in deze bachelorproef meer over dit onderwerp te weten komt en nieuwe kennis opdoet.

Samenvatting

In deze bachelorproef wordt een vergelijkende studie uitgevoerd tussen een mainframe computer (z15) en een computer met x86-architectuur. De vergelijkende studie draait om de prestatie verschillen tussen de twee systemen. Om deze specifiek te houden, wordt onderzocht welke systemen Java-code beter/sneller kan uitvoeren.

Het voornaamste doel van deze bachelorproef is om uit te zoeken welk computersysteem beter presteert bij het uitvoeren van dezelfde Java-code. De doelgroep van dit onderzoek zijn daarom gebruikers van Java-code op zowel mainframes als andere op X86-architectuur gebaseerde systemen. Er wordt verwacht dat de testen een beeld kan geven van de prestatie verschillen waaruit een beperkte zicht kan worden verkregen voor gebruikers die overwegen om over te stappen naar een ander platform.
Om te beginnen werden beide computersystemen grondig onderzocht. Vervolgens werd de gebruikte programmeertaal onderzocht en verschillende kenmerken besproken. Daarnaast werden de algoritmen die in dit onderzoek werden gebruikt onderzocht en werd er een diepgaande uitleg geschreven over deze algoritmen.
Na dit uitgebreide onderzoek werden de codes op beide systemen uitgevoerd met verschillende invoer parameters. Voor het algoritme van Fibonacci werden de getallen tussen 50 en 57 getest op een niet-presterende recursieve manier. Voor priemgetallen werd in eerste instantie de Algoritmen Zeef van Eratostenes als proef deling getest.

Uit de uitgevoerde testen, waarvan de resultaten te vinden zijn in het hoofdstuk Resultaten, blijkt dat het mainframe systeem beter presteert .
Maar deze uitgevoerde testen zijn niet genoeg en er zouden veel meer moeten worden uitgevoerd. De systemen waarop de zijn uitgevoerd hebben geen vergelijkbare gebruiksdoeleinden dus de testen zouden moeten worden uitgevoerd op vergelijkbare systemen. Het zou representatiever zijn geweest als de waren uitgevoerd op een X86-gebaseerde server dan op een persoonlijke laptop.

Uit de resultaten van de verrichte onderzoeken kan worden afgeleid dat de prestaties voor het berekenen van priemgetallen de mainframe beter presteert op tijd als geheugen. Maar voor de fibonaci algoritme is het ook zichtbaar dat de x86 sys-

teem een kortere uitvoer tijd als CPU tijd hebben vergeleken met de mainframe maar waarvan op vlak van geheugen presteert mainframe veel beter.

Er kan worden geconcludeerd dat de resultaten niet representatief zijn voor het vergelijken van de prestaties van beide systemen.

Inhoudsopgave

Lijst van figuren

1

Inleiding

1.1. Probleemstelling

In de tijd waarin organisaties streven naar het optimaliseren van hun IT toepassingen en zo veel mogelijk winst wensen te maken rijst de vraag welke platform de beste keuze is. Mainframe of moderne x86 gebaseerde systemen?

Hoewel mainframes systemen bekend staan om hun krachtige hardware en hoge betrouwbaarheid, zijn moderne pc 's de afgelopen jaren sterk geëvolueerd en bieden ze indrukwekkende prestaties tegen lagere kosten.

In deze onderzoek zal er specifiek gefocust worden op de prestatie verschillen van het uitvoeren van java code op beide systemen.Daarnaast ligt de focus op welke mogelijke voordelen het gebruik van java biedt.

1.2. Onderzoeksvraag

De vragen van deze onderzoek zijn als volgt:

- Wat zijn de prestatie verschillen tussen Java-toepassingen die op mainframes en moderne pc 's worden uitgevoerd voor verschillende typen computationeel intensieve taken?
- Waarom zou het interessant zijn om in te zetten in mainframe systemen ?
- Vanwaar de interesse voor X86-systemen?
- Welke voordelen brengt het gebruik van Java met zich mee op mainframe systemen?

1.3. Onderzoeksdoelstelling

In het kader van dit onderzoek is het doel om eventuele prestatie verschillen tussen mainframe computers en computersystemen die gebaseerd zijn op de x86 processor architectuur vast te stellen.

Om dit te realiseren zullen er java code geschreven worden waar dat er reken intensieve algoritmen in gebruikt wordt. Dit om werklast te simuleren.
Tijdens het uitvoeren van de programma's zullen de volgende zaken bijgehouden om systemen met elkaar te kunnen vergelijken:

- De totale tijd die nodig is om code uit te voeren
- De CPU-tijd die het programma
- en het totale geheugengebruik.

1.4. Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:
In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.
In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoektechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.
In Hoofdstuk 4 wordt de resultaten van de testen visueel getoond
In Hoofdstuk 5 Wordt de gebruikte code's om testen uit te voeren getoond
In Hoofdstuk 6, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2

Stand van zaken

2.1. Java

2.1.1. Geschiedenis van Java

In deze deel van de tekst zullen we het hebben over hoe dat Java tot stand is gekomen.

Eerst en vooral gaan we het hebben over hoe dat de programmeertaal dat we op de dag van vandaag kennen als Java tot stand is gekomen. Hiervoor gaan we terug in de tijd tot de jaren negentig van de voorbije eeuw.

Java was ontworpen in een vrij klein development team dat in eerste instantie het ”Stealth Project”werd genoemd. Later werd dit team groter en kreeg het de naam ”Project Green”. Verschillende bronnen hebben verschillende redenen genoemd voor het ontstaan van Java. Hier zullen we het enkel hebben over twee redenen.

Volgens wat in het boek **java 17 for absoulte begginers** Cosmina, 2021b wordt vermeld, was de reden dat een Amerikaans bedrijf dat de revolutie in de computerwereld leidde, Sun Microsystems, besloot zijn beste ingenieurs te verzamelen om een product te ontwerpen en te ontwikkelen dat het bedrijf in staat zou stellen een belangrijke speler te worden in de nieuwe opkomende internetwereld (Cosmina, 2021b).

Volgens een andere bron is de reden voor de ontwikkeling anders. Namelijk dat het oorspronkelijke project met de codenaam Green bedoeld was om ”slimmeëlektronische apparaten voor consumenten te ontwikkelen (zoals TV top controle boxen). Er was software nodig om deze kleine, potentieel slecht presterende maar gevarieerde apparaten te besturen. Het Green team wilde geen C of C++ gebruiken van-

wege technische problemen met deze talen (niet in de laatste plaats portabiliteits problemen). Ze besloten daarom hun eigen taal te ontwikkelen en in augustus 1991 was een nieuwe objectgeoriënteerde taal geboren (Hunt, 2013).

Om diverse redenen werd deze programmeertaal in ontwikkeling gebracht, andere redenen zullen hier niet behandeld worden. Maar het duurde tot 1996 voordat de eerste stabiele versie van Java op de markt werd gebracht. Vanwege juridische redenen was de uitgave uitgesteld op het genoemde jaar. Omwille van de naam die er oorspronkelijk aan gegeven was, ontstonden er problemen.

De programmeer taal die we op de dag van vandaag kennen als Java werd toen met een andere naam vernoemd. Tijdens de ontwikkeling werd Java oorspronkelijk Oak genoemd, daarna hernoemd naar Green, en uiteindelijk uitgebracht als Java van Java-koffie, vandaar dat het logo voor Java een dampende koffiekop is (Winnie, 2021)..

De reden waarom het niet **Oak** mocht genoemd worden, was vanwege het feit dat die naam al het handelsmerk was van Oak Technologies(Krill, 2022).

Oak's logo werd later gebruikt voor een interactieve software agent genaamd Duke. Duke was de interactieve host die een nieuw soort gebruikersinterface mogelijk maakte die meer was dan de knoppen, muizen en pop-up menu 's van de desktop computerwereld. Tegenwoordig wordt Oak beschouwd als de mascotte van Java. (Oracle Corporation, 2024)

In de daaropvolgende jaren na de introductie van de taal op de markt bleef de populariteit alsmaar stijgen. In het derde kwartaal van het jaar 2001 werd het de populairste programmeertaal en het behield zijn positie tot het eerste kwartaal van 2019. Bij het schrijven van dit onderzoek is het nog steeds een van de populairste programmeertalen die in gebruik Het wordt op het moment door verschillende sectoren en platformen gebruikt zoals mobiele platforms, IOT, ... (Statistics & Data, 2023)

Een van de redenen waarom het zo populair is geworden, waar er in het volgende hoofdstuk meer in detail zal worden gesproken , is het feit dat de werking gebaseerd is op de "Write once, run anywhere"logica in het Nederlands "schrijf het een keer, voer het overal uit". Dit betekende dat programma's die in Java werden gemaakt maar één keer gecodeerd hoefden te worden en op elk platform konden draaien dat Java ondersteunde(Winnie, 2021).

Apart van deze feit was de manier van uitgave ook een redelijk belangrijk aspect

waarom het zo populair werd. De ontwikkelaars kozen ervoor om de programmeertaal toegankelijk te maken via het internet. Door deze manier van publicatie was het mogelijk om de taal te verspreiden onder een breed publiek, van waaruit een algemene acceptatie plaatsvond. Op 20 april 2009 hebben de bedrijven Oracle Coporation end Sun Microsystems het aangekondigd dat ze een definitieve overeenkomst hebben gesloten waarbij Oracle gemeenschappelijke Sun-aandelen zal overnemen voor $9,50 per aandeel in cash(Oracle Corporation, 2009).
Hierdoor werd onder andere de programmeertaal Java eigendom van Oracle. Na deze overname is Oracle de development rond Java beginnen ondersteunen en zijn er al verschillende versies uitgekomen onder Oracle.

2.1.2. JVM: Java Virtual MachineJava Virtual Machine

JVM is een cruciaal onderdeel bij het gebruiken, ontwerpen, uitvoeren en testen van applicaties die geschreven zijn in Java. De JVM is het onderdeel dat ervoor zorgt dat de geschreven code kan uitgevoerd worden. Apart hiervan is het ook het vertaal kanaal voor de machine om Java code om te zetten in machinetaal zodat de apparatuur waar op de code moet uitgevoerd worden de code kan interpreteren en kan uitvoeren.

Hierdoor kunnen Java-toepassingen platformonafhankelijke code uitvoeren op verschillende apparaten met verschillende specificaties en architectuur zonder dat de code hoeft te worden gewijzigd. Java programma's kunnen dit doen in tegenstelling tot andere programmeer talen zoal C of C++. Dit allemaal door gebruik te maken van een virtuele machine die boven op de hardware en besturingssysteem werkt. Kort gezegd is het een abstracte computermachine waarmee een computer een Java-programma kan uitvoeren(Cosmina, 2021b).

Zoals eerder vermeld is de werking van de JVM verschillend dan vergelijkbare programeer talen zoals C. De werking is als volgt:
De Java Virtuele Machine weet niets van de programmeertaal Java, het weet alleen iets van een bepaalde binair formaat, het class bestandsformaat. Een klas bestand bevat Java Virtuele Machine instructies (of bytecodes) en een symbool tabel, evenals andere aanvullende informatie. (Lindholm, 2013)
Bovendien is het ook een tool die beveiliging biedt voor : Omwille van de veiligheids redenen legt de Java Virtuele Machine sterke syntactische en structurele beperkingen op aan de code in een klassebestand(Lindholm, 2013).

Java is ook niet het enige programmeertaal dat gebruik kan maken van de JVM andere programmeertalen zoals Groovy, Scala, Kotlin en Clojure zijn allemaal erg populaire programmeertalen die draaien op de JVM(Cosmina, 2021a).

Het gebruik maken van een JVM heeft ook zijn nadelen. Omwille van de extra compilatiestap die genomen moet worden om de geschreven code over te zetten naar machinetaal. zodat het geïnterpreteerd kan worden door de computer die het programma uitvoert. Maakt het in vergelijking met puur gecompileerde taal zoal C++ trager bij het uitvoeren(Cosmina, 2021a).

Ook is het belangrijk te vermelden dat door het gebruik van de JVM het naast platform en besturingssysteem onafhankelijkheid, het een kleine omvang heeft van de gecompileerde code en het de eigenschap heeft om gebruikers te beschermen tegen kwaadaardige programma's(Lindholm, 2013).

Het is ook belangrijk om te vermelden dat programeer talen van niveau kunnen verschillen. Sommige programmeertalen zijn high level en sommige low level. Het verschil tussen deze twee niveaus zijn voornamelijk dat de lower level programeer talen dichter bij de hardware liggen in vergelijking met de high level programeer talen. In tegenstelling tot Assembler is Java een High level programeer taal. Dit betekent dat het geen programeer taal is dat dicht bij de hardware zit, in tegendeel ligt er een extra laag tussen namelijk de Java Virtuele machine.

Dit is ook wat Java apart zet van andere high level programeer talen, is dat het een extra laag heeft boven op de besturingssysteem en hardware. Deze laag krijgt de high level geschreven code en compileert het naar een soort tussentaal dat redelijk dichtbij machine code light namelijk java-bytcode. Deze java-bytcode wordt gecompileerd voor bepaalde hardware, om specifieker te zijn de processor. Aangezien de architecturen van processoren kunnen verschillen, zoals x86 x64, arm of andere processor architecturen, wordt het specifiek gecompileerd voor de gebruikte processor. Hierdoor kunnen toepassingen die op JVM draaien platformonafhankelijk functioneren.

JVM is kortom is een abstracte rekenmachine waarmee een computer een Java-programma kan uitvoeren. Het is een platform-onafhankelijke executie-omgeving die Java-code omzet in machinetaal en deze uitvoert(Cosmina, 2021a).

2.1.3. JIT : Just-In-Time

JIT is een compiler dat onderdeel is van de uitvoer omgeving van Java en staat voor Just-In-Time.

Als standaard is de JIT-compiler actief maar dit kan ook manueel uitgeschakeld worden. Dit wordt echter niet geadviseerd vanwege de slechte gevolgen voor de

prestaties. Behalve als we het hebben over JIT compilatie problemen, om deze te kunnen identificeren of er omheen te werken zou uitschakelen kunnen helpen (IBM, 2024).

Zoals eerder vermeld wordt er tijdens de uitvoering van een programma code omgezet naar een taal die door de computer interpreteer baar is. Deze Java-programma's zijn opgebouwd uit klassen die platform-neutrale bytecodes bevatten die kunnen worden geïnterpreteerd door een JVM op veel verschillende computer architecturen. Tijdens het uitvoeren laadt de JVM de klasse bestanden, deze bepalen de semantiek van elke individuele bytecode en voert de juiste berekeningen uit (IBM, 2024).

Een nadeel van JIT is dat het systeembronnen inmeet en dus ook tijd. Vanwage de JIT-compilatie vereist het processortijd en geheugen gebruik. Wanneer de JVM voor het eerst opstart, worden duizenden methoden aangeroepen. Het compileren van al deze methodes kan de opstarttijd aanzienlijk beïnvloeden, zelfs als het programma uiteindelijk zeer goede piekprestaties bereikt (IBM, 2024).

Om de prestatie te verbeteren is het mogelijk om JIT in te stellen in verschillende optimalisatie niveau. Deze zijn namelijk de niveaus: koud, warm heet, zeer heet of verzengend (IBM, 2024).

Het standaard is dit ingesteld op "warm", maar dit kan worden aangepast door de ontwikkelaars. Een hoger niveau van optimalisatie resulteert in betere prestaties met betrekking tot geheugengebruik en processorbelasting, maar in tegendeel gaat dit ten koste van de prestatie wanneer het niveau wordt verlaagd. Het is van belang op te merken dat bij een "cold" optimalisatieniveau de opstarttijd voordeliger kan zijn.

2.1.4. Garbage collection

Garbage collection is zoals JIT een componenten van de Java platform. De functie van deze component is dat het automatisch het geheugen beheer regelt.

Het gebruik van de garbage collector is belangrijk voor applicaties en het systeem waarop ze draaien. De voordelen zijn onder andere Geheugenoptimalisatie, Voorkomen van memory leaks, Betere prestaties, Geen handmatig geheugenbeheer en dus Dynamische geheugentoewijzing.

2.1.5. JDE: Java Development Environment

Zoals de naam al aanduidt, is het de omgeving waarin het ontwikkelen van code wordt uitgevoerd. Het is ontworpen met als doel de ontwikkelaars een omgeving te voorzien waar dat het mogelijk is om code te schrijven, te compileren en te debuggen/testen. Het is in essentie een verzameling van de nodige componenten en hulpmiddelen om Java applicaties te schrijven. Een van de belangrijkste componenten van JDE is Java Development Kit (JDK). Dit is kortom een software pakket met de nodige middelen om Java code te schrijven , meer informatie hierover wordt in detail behandeld in het hoofdstuk "Java Development Kit".

JDE kan soms verward worden met Integrated Development Environment (IDE), waartussen het verschil tussen de twee niet zo groot is. In essentie is de JDE een gespecialiseerd IDE, meer specifiek het is een IDE geoptimaliseerd voor het ontwikkelen van java-code.

Een van de meest gekende voorbeeld van een ontwikkelingsomgeving geoptimaliseerd voor Java is Eclipse. Apart van Eclipse zijn er veel meer andere Development environments waar er code kan ontwikkeld worden zoals NetBeans, InelliJ IDEA en andere.

Tenslotte is het niet nodig om een JDE of een IDE te gebruiken om Java code te schrijven, het enigste dat nodig is, is de Java Development Kit om de code uit te voeren. Dus het is perfect mogelijk om Java code te schrijven in een tekst editor zoals het vermeld wordt in het boek:" (Cosmina, 2021a).

2.1.6. JRE: Java Runtime Environment

Zoals het naam het aangeeft is het de omgeving waar dat Java applicaties uitgevoerd worden op een computer. Om Java-programma's uit te voeren heeft de JRE enkele zaken nodig. Het heeft Ten eerste de Java Virtual Machine die zoals eerder vermeld, om het uitvoer van java-bytcode regelt. Daarnaast heeft het de Java Class Libraries nodig.

Hoe dat het in zijn werk gaat wordt als volgt uitgelegd. De JRE combineert Java-code gemaakt met behulp van de JDK met de nodige bibliotheken die nodig zijn om het op een JVM te draaien en maakt dan een instantie van de JVM aan die het uiteindelijke programma draait („What is the Java Runtime Environment (JRE)? ", 2021).

2.1.7. JDK: Java Development Kit

Zoals de naam aanwijst is dit een pakket voor Java ontwikkeling. Deze pakket bevat de Java compiler die de Java code omzet naar bytecode zodat het door de JVM kan uitgevoerd kan worden.

Het bevat ook de JVM zelf om de bytcode uit te voeren op de host hardware. Daarnaast bevat het ook de JRE die verantwoordelijk is voor het uitvoer van de code. Ook heeft het de nodige hulpmiddelen en classes dat gebruikt wordt tijdens het uitvoer van de code.

Kortom is het een gehele verzameling van de nodige bibliotheken en tools die gebruikt worden door software ontwikkelaars om Java Applicaties te maken, te compileren, te testen en ook uit te voeren.

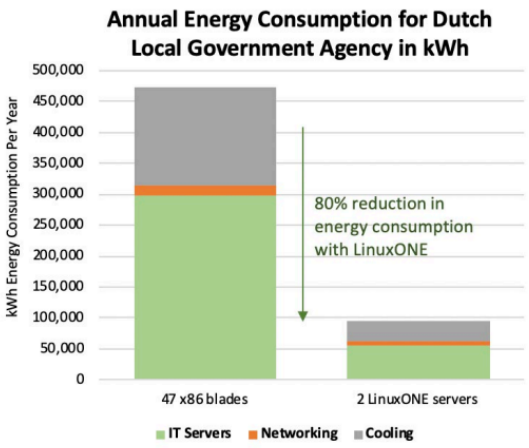
2.2. Mainframe

2.2.1. Waarom mainframe?

Mainframe computers hebben verschillende eigenschappen die ervoor zorgen dat ze tegenwoordig nog steeds interessant kunnen zijn.
Een daar van is de ecologische voetafdruk dat veel lager is in vergelijking tot de x86 architectuur gebaseerde servers. Een voorbeeld dat aantoont dat mainframe computers ecologische beter zijn kan gevonden worden in Nederland.
Om te voldoen aan het besluit van het Nederlandse Hoog raad om de uitstoot van broeikasgassen met 25% te verminderen ten opzichte van het niveau van 1990 tegen het einde van 2020, heeft een Nederlandse lokale overheidsorganisatie een deel van haar x86 server IT-infrastructuur door een IBM LinuxONE™ omgeving vervangen. Door de Linux-applicaties te verplaatsen van 47x86 blades naar 11 IFL's op LinuxONE, zal het energieverbruik voor de organisatie naar verwachting met 80% afnemen, wat resulteert in 946 ton minder CO2-uitstoot over een periode van vijf jaar (Donohue, 2020).

In Grootschalige bedrijfstoepassingen waar meerdere x86-servers worden gebruikt en er behoefte is aan zowel een grote fysieke ruimte als de bijbehorende benodigdheden om deze systemen te laten draaien zoals koeling, elektriciteit en de benodigde mankracht om deze te onderhouden, kunnen de kosten sterk toenemen.

Dit was ook het geval in een grote verzekeringsmaatschappij in Azië Pacific. Deze organizatie maakte een aanzienlijke grote IT groei. Naarmate het x86-datacenter groeide, namen de fysieke vloeroppervlakte en hardwarekosten toe, de energierekeningen hoger en worstelde het IT-personeel met de complexiteit van het server-



Figuur (2.1)
Jaarlijks energieverbruik voor Nederlandse lokale overheidsinstantie in kWh
Donohue, 2020

beheer.

Het bedrijf startte een evaluatie van de huidige activiteiten om effectiever te kunnen schalen voor nieuwe werklasten. Uit analyse bleek dat werklasten die op 55 x86-servers draaiden, geconsolideerd konden worden op één IBM LinuxONE-systeem met een drastische afname in energie- en vloergebruik. Het vloeroppervlak kon met 86% worden gereduceerd en het jaarlijkse energiegebruik kan met 62% dalen. Deze besparingen stellen het bedrijf in staat om hun uitdagingen van snelle groei aan te gaan met veel dichtere werkbelasting, een kleiner datacenter en eenvoudiger beheer voor het personeel (Donohue, 2020).

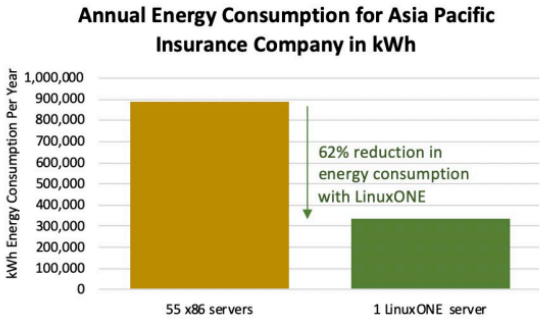
Zoals u kunt zien op bovenstaande voorbeelden is het duidelijk dat het ecologisch voetafdruk van mainframe computers veel lager ligt dan zijn tegen hangers. Bovendien bieden deze mainframe computers geavanceerde eigenschappen zoals schaalbaarheid, beveiliging, betrouwbaarheid, kwantum safe en meer. Het is wel nodig om te vermelden dat mainframe computers voor kleinschalige toepassingen een overbodige investering kan zijn vanwege het hoge prijskaartje dat eraan hangt.

2.2.2. Waarom Java op de mainframe?

Een van de hoofdzakelijke reden om Java te gebruiken op de mainframe is wegens de reden dat het veel gemakkelijker is om mensen te vinden die met deze programmeer taal kunnen werken, in vergelijking met mainframe specifieke programmeertalen zoals Cobol en PLI. Bovendien is het ook een object georiënteerde taal in vergelijking met de andere talen die gebruikt worden op het mainframe.

x86 and LinuxONE Energy and Floor Space Comparisons for Asia Pacific Insurance Company

Data Center Requirements	x86	LinuxONE	Savings
Energy	890,016 kWh	335,508 kWh	62%
Floor space	42.57 meters	6.11 meters	86%



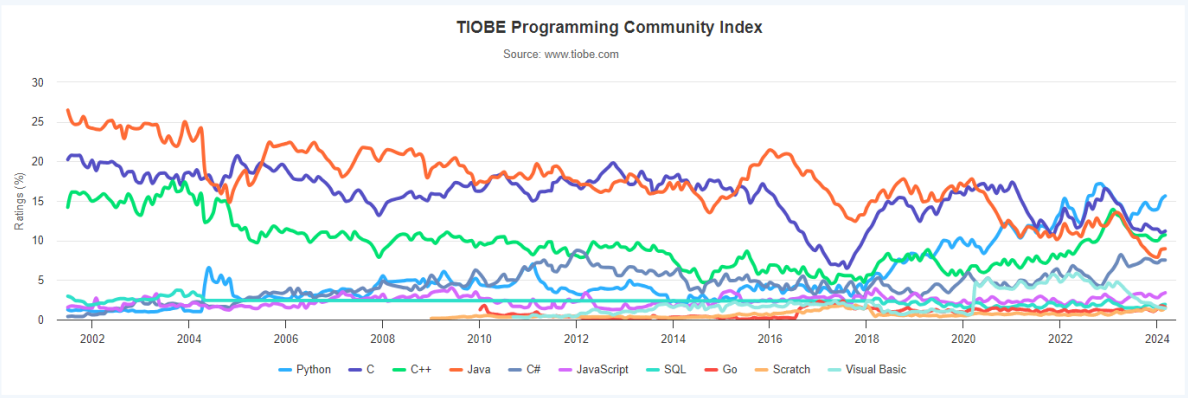
Figuur (2.2)
Jaarlijks energieverbruik voor asia pacific verzekeringsmaatschappij in kWh

Donohue, 2020

Dit maakt het mogelijk dat software ontwikkelaars herbruikbare en modulaire code kunnen schrijven.
Volgens de objectgeoriënteerde theorie is een oplossing op basis van objecten op de lange termijn goedkoper en gemakkelijker te onderhouden (Byrne & Cross, 2009).

Een van de meest bekende en belangrijkste aspecten en wat het uniek maakt vergeleken met andere programeertalen zoals C of C++ is dat Java Applicaties platform onafhankelijk werken. Dit betekent dat de Java code gemakkelijker van een computer systeem naar een andere computer systeem kan uitgevoerd worden. Dit kan er voor zorgen dat taken die niet hoeven te draaien op de mainframe later gemakkelijker kunne worden overgezet naar een andere systeem in vergelijking tot de andere programmeertalen zoals Cobol of PLI. Voor een meer uitgebreid uitleg over de platform onafhankelijkheid kunt u de java sectie van dit paper raadplegen.

Apart van bovenstaand gegevens is Java nog steeds één van de meest populaire programmeer talen, tegenwoordig bevind Java zich ongeveer in de top 5 met de andere populaire talen Python, C, C++ en C#.



Figuur (2.3)
The TIOBE Programming Community index
(„TIOBE Index for May 2024", 2024)

Wegens de reden dat het een van de meest gebruikte programmeertalen is, is het een aantrekkelijker optie omdat er veel meer ontwikkelaars te vinden zijn. In vergelijking met de gebruikte talen op mainframe systemen zoals PLI en COBOL zijn er veel meer ontwikkelaars die Java applicaties schrijven. Er moet ook voor ogen gehouden worden dat er hier besproken wordt over de populariteit en niet over hoe dat ze presteren, dit is een helemaal andere topic op zich.

Bovenstaand afbeelding toont aan dat het nog een relevante en veelal gebruikte programeer taal is. Dit maakt het interessant om Java te gebruiken om bijvoorbeeld legacy-code te vervangen met een modernere porgrameer taal. Vele organisaties die mainframes draaiend hebben, gebruiken legacy systemen en bedrijfskritieke applicaties.
Meestal zijn deze code jaren geleden geschreven en werken zoals de eerste dag zonder problemen. Dit komt door dat de mainframe computers achterwaarts compatibel zijn. Maar wanneer er nood is aan verandering kunnen er problemen of belemmeringen ontstaan. Door deze belemmeringen die zouden kunnen op treden tegen te gaan en ook de code te moderniseren kan Java gebruikt worden (IBM Corporation, 2023).

2.2.3. z15

In dit hoofdstuk wordt met de nadruk gelegd op de mainframe computer waarop onder andere het onderzoek uitgevoerd zal worden. Het is echter niet overbodig om te vermelden dat inmiddels de voorloper z16-series systemen al uit en in gebruik zijn met diverse extra functionaliteiten ten opzichte van de voorgaande z15-series systemen. Aan deze systemen zullen we hier echter niet veel aandacht besteden, aangezien het niet het systeem is waarop we werken . We zullen in deze sectie kort de algemeen gekende aspecten van mainframe computers en z15 be-

spreken.

De IBM Z15-serie mainframe systemen is een mainframe computer geproduceerd door Tech Giant IBM als onderdeel van de Z-serie mainframe computers. Deze computer is een schaalbare en krachtige bedrijfscomputer die speciaal is gemaakt voor bedrijf kritische toepassingen.

Een van de bekendste eigenschappen van de mainframe computers is de gegarandeerde beschikbaarheid die ze kunnen bieden aan hun klanten. In de documentatie van IBM over de beschikbaarheid van hun systemen wordt het volgende vermeld hierover:

Het betrouwbaarheids- en redundantieniveau dat in mainframes wordt geïntroduceerd ligt in de orde van 99,999% (de vijf 9's), waardoor er nog steeds ongeplande uitval is van ongeveer 5,3 minuten per jaar. Om een nog hogere beschikbaarheid te bereiken, introduceerde IBM een clusteringtechnologie met de naam Parallel Sysplex (IBM Corporation, 2010)

De Z15 computer en verschillende andere versies van de Z-serie mainframe computers staan ook bekend om hun data security, hun indrukwekkende prestaties, schaalbaarheid, betrouwbaarheid en achterwaarts compatibiliteit.

Vooral voor het laatste aspect, namelijk achterwaartse compatibiliteit, geldt dat programma's die geschreven zijn op de systemen S/360 uit de jaren 60 nog steeds uitvoerbaar zijn op de allernieuwste z16 mainframe machines vanwege de nadruk op achterwaartse compatibiliteit.

Naast deze kenmerken heeft het zoals eerder vermeld in het hoofdstuk over "Waarom mainframe?" een kleinere ecologische voetafdruk voor de grootschalige toepassingen die er op draaien in vergelijking met systemen waarmee het vergeleken wordt, zoals clustercomputers.

Hardware informatie

Omwille van de confidentialiteit wordt er geen specifieke systeeminformatie gedeeld. Algemene informatie betreft z15-systemen komen in dit hoofdstuk wel aan bod.

Z15 systemen kunnen in verschillende formaten en eigenschappen geconfigureerd worden. De footprint voor de z15 is gebouwd met een 19-inch formaat dat 1 - 4 frames schaalt voor de z15 T01 afhankelijk van de configuratie, en een enkel frame voor de z15 T02 (IBM Corporation, 2010).

Figuur (2.4)
z15 series mainfraime computer

(IBM Corporation, g.d.-a)

De z15 series mainframe systemen zijn behoefte schaalbare systemen, de architectuur zorgt voor continuïteit en upgradebaarheid ten opzichte van de vorige z14 en IBM z13® modellen. Het z15 T01 model heeft vijf bestelbare functies: Max34, Max71, Max108, Max145 en Max190.Het z15 T02 model heeft vijf bestelbare functies: Max4, Max13, Max21, Max31 en Max65(IBM Corporation, g.d.-a).
Hieronder worden de technische eigenschappen in een overzichtelijke tabel weergegeven.

Tabel 2.1: Technical highlights of the z15

Mogelijkheden	z15 TO1	z15 T02
Grotere totale systeemcapaciteit en meer subcapaciteit instellingen voor CP's. De IBM z/Architectuur zorgt voor continuïteit en upgradebaarheid ten opzichte van voorgaande modellen.	Tot 190 karakteriseerbare PU's. Tot 34 CP's ondersteunen 292 subcapaciteitsinstellingen.	Tot 65 karakteriseerbare PU's. Tot 6 CP's met 156 instellingen voor subcapaciteit.
Multi-core, single-chip modules (SCM's) die draaien om de uitvoering te helpen optimaliseren van processor-intensieve werklasten.	5.2GHz (14 nm FINFET Silicon-On-Insulator [SOII).	4.5GHz (14 nm FINFET Silicon-On-Insulator [SOI]).
Meer werkelijk geheugen per systeem, dat zorgt voor een hoge beschikbaarheid in het geheugen-subsysteem door gebruik te maken van bewezen redundante array van onafhankelijk geheugen (RAIM) technologie.	Tot 40 TB adresseerbaar werkelijk geheugen per systeem.	Tot 16 TB adresseerbaar werkelijk geheugen per systeem.

(Redbooks, 2020)

Tabel 2.2: Table 1-2 Technical highlights of the z15

Mogelijkheden	z15 T01	z15 T02
Een groot vast hardware systeemgebied (HSA) dat apart van het door de klant gekochte geheugen wordt beheerd.	256 GB.	160 GB.
Bewezen technologie (vijfde generatie hoge frequentie en derde generatie out-of-order ontwerp) met een single-instruction, multiple-data (SIMD) processor die het parallellisme verhoogt om de verwerking van analyses te versnellen. Daarnaast kan simultaan multithreading (SMT) de verwerkingsefficiëntie en verwerkingscapaciteit verhogen en het aantal lopende instructies verhogen. Speciale co-processors en nieuwe hardware-instructies voor het versnellen van geselecteerde werklasten.		
Processor cache structuur verbeteringen en grotere cache groottes om te helpen met de hedendaagse veeleisende productie workloads. Beide z15-modellen hebben de volgende cache-niveaus: →Eerste laag cache (L1 privé): 128 KB voor instructies, 128 KB voor data →Tweede laag cache (L2): 4 MB voor instructies en 4 MB voor data. →Derde laag cache (L3): 256 MB →Vierde-laag-cache (L4): 960 MB		
Verbeterde cryptografische functies en prestaties ten opzichte van hun voorgangers, wat wordt bereikt door één speciale cryptografische co-processor per PU. Nieuwe cryptografische mogelijkheden met Crypto Express7S, zoals beveiligde sleutelverbeteringen voor sleutelbeheer, aanmaken van digitale handtekeningen en verificatie met nieuwe sleutels en algoritmen.		
zHyperLink biedt een verbinding met lage latentie met opslag-subsystemen voor het sneller ophalen van gegevens. Deze functie is consistent met hun voorgangers.		
Vergeleken met hun voorgangers worden verbeterde gegevenscompressiebewerkingen bereikt door een speciale Compressie Co-processor per PU en nieuwe hardware-instructies. Daarnaast biedt de On-Chip compressieversneller een per-chip vervanging voor de IBM zEnterprise® Data Compression (zEDC) Express functie.		
Het channel-subsysteem (CSS) is gebouwd voor I/O-veerkracht. Het aantal logische kanaalsubsystemen (LCSSs), subkanaalsets en I/O-apparaten is consistent met het vorige platform, net als het aantal logische partities (LPARs).	→Sez LCSS →85 LPARs →vier subkanalen sets →64,000 I/O apparaten per set subkanalen	→Drie LCSS →40 LPAR →Drie subkanalen sets →64,000 I/O apparaten per set subkanalen

(Redbooks, 2020)

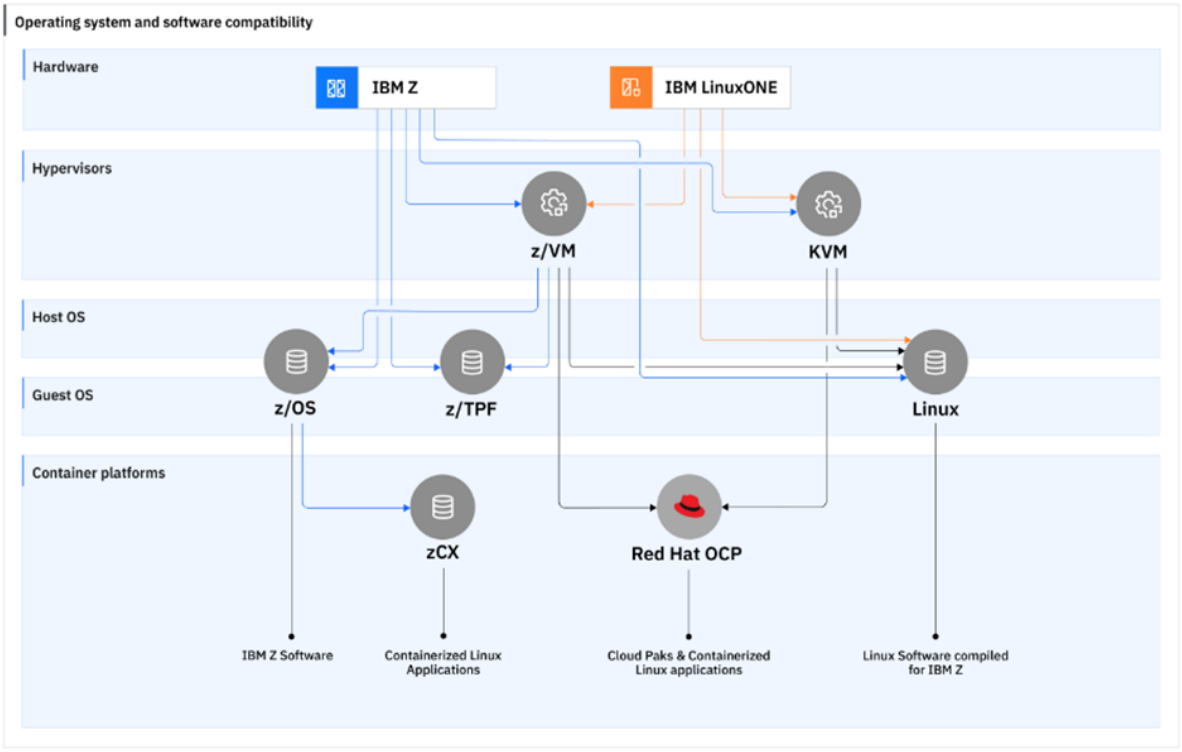
2.2.4. Besturingssystemen

Voordat er meer wordt verteld over de verschillende typen processors die op mainframe computers zijn geïnstalleerd, of beter gezegd over processors met verschillende karakteristieken, waar uitgebreide informatie te vinden is in de volgende 2 hoofdstukken, namelijk "Central Processor Complexën "Specialty Processors (SAP)".

Wordt eerst besproken wat een besturingssysteem is en vervolgens welke besturingssystemen draaien op deze mainframe systemen. Volgens de Universiety of Wollongong Austrralio is het besturingssysteem (OS) verantwoordelijk voor het beheren van zowel de software als de hardware van een computer. Het voert basistaken uit zoals bestands-, geheugen- en procesbeheer, het afhandelen van invoer en uitvoer en het aansturen van randapparatuur zoals schijfstations en printers (University of Wollongong, 2024).

Het is ook belangerijk te vermelde dat besturingssystemen er op zijn gericht om optimaal gebruik te maken van de verschillende bronnen van de computer en ervoor te zorgen dat de maximale hoeveelheid werk zo efficient mogelijk wordt verwerkt. Hoewel een besturingssysteem de snelheid van een computer niet kan verhogen, kan het wel het gebruik van bronnen maximaliseren, waardoor de computer sneller lijkt te werken doordat het meer werk kan uitvoeren in een bepaalde tijd(IBM, 2023d).

De meest bekende en populaire besturingssysteem dat op mainframe systemen draaien is z/OS. Naast z/OS domineren vier andere besturingssystemen het mainframegebruik: z/VM, z/VSE™, Linux voor System z® en z/TPF, waarin elk besturingssysteem hieronder kort wordt besproken(IBM, 2023d).



Figuur (2.5)
Mainframe Besturingssystemen

(IBM, 2023c)

1. **z/OS**
   z/OS® is een veelgebruikt mainframe besturingssysteem ontworpen om een stabiele, veilige en continu beschikbare omgeving te bieden voor applicaties die op het mainframe draaien. Deze applicaties zijn voornamelijk bedrijfscrititeke applicaties waar dat deze eigenschappen van crusiaal belang zijn(IBM, 2023a).

2. **z/VM**
   z/Virtual Machine (z/VM®) is een hypervisor doordat het andere besturingssystemen draait in de virtuele machines die het aanmaakt(IBM, 2023b).

   IBM z/VM is ontworpen om honderden tot duizenden gastservers te draaien op een enkele IBM Z server of IBM LinuxONE server (waar enkel Linux en z/VM gasten ondersteund worden) met de hoogste graad van efficiëntie en elasticiteit. Een fundamentele kracht van z/VM is de mogelijkheid voor virtuele machines om systeembronnen te delen met zeer hoge niveaus van resourcegebruik. z/VM biedt extreme schaalbaarheid, beveiliging en efficiëntie om mogelijkheden te creëren voor kostenbesparingen, terwijl het een robuuste basis biedt voor cognitive computing op de IBM Z en LinuxONE platformen (IBM, 2023f).

3. **z/VSE**
   z/Virtual Storage Extended (z/VSE™) is populair bij gebruikers van kleinere mainframe computers. Wanneer gebruikers van z/VSE de vereiste capaciteiten ontgroeid zijn, hebben ze ook de mogelijkheid om over te stappen naar z/OS® waar de capaciteiten veel hoger liggen. Er zou een vergelijking kunnen worden gemaakt over een soort Lite versie van z/OS(IBM, g.d.-d).

   Vergeleken met z/OS biedt het z/VSE besturingssysteem een kleinere, minder complexe basis voor batchverwerking en transactieverwerking. Het ontwerp en de beheerstructuur van z/VSE is uitstekend voor het draaien van routinematige productiewerklasten die bestaan uit meerdere batch jobs (die parallel draaien) en uitgebreide, traditionele transactieverwerking(IBM, g.d.-d).

4. **Linux on IBM Systems**
   Het enterprise Linux besturingssysteem is een solide basis voor uw open-source, hybride cloud-infrastructuur. Het draaien van Linux op IBM servers brengt een nieuw niveau van betrouwbaarheid, beveiliging en schaalbaarheid voor bedrijfskritische workloads. Verminder de complexiteit en verbeter de prestaties met Linux op IBM Z®, IBM® Linux ONE en Linux op IBM Power®(IBM, g.d.-b).

5. **z/TPF**
   Het IBM z/Transaction Processing Facility (z/TPF) systeem is een besturingssysteem met hoge beschikbaarheid, ontworpen om snelle reactietijden te bieden voor grote hoeveelheden berichten van grote netwerken van terminals en

werkstations(IBM, g.d.-c).

z/TPF is een systeem bedoeld voor speciale doeleinden dat wordt gebruikt door bedrijven met een zeer hoog transactievolume, zoals creditcardbedrijven en reserveringssystemen voor luchtvaartmaatschappijen(IBM, g.d.-e).

2.2.5. Processoren

Central Processor Complex

De Central Processor Complex (CPC) staat voor het concept om meerdere processors tegelijk te gebruiken in plaats van te moeten rekenen op slechts één processor.

Zoals het in de IBM documentatie wordt vermeld was dit niet altijd het geval. Vroege mainframes hadden één processor, die bekend stond als de centrale processor (CPU). De huidige IBM® mainframes hebben een centraal processorcomplex (CPC), dat verschillende typen z/Architecture® processors kan bevatten die voor verschillende doeleinden kunnen worden gebruikt. (IBM Corporation, g.d.-b).

Deze processoren hebben ook een aantal andere functies, waarvan sommige te maken hebben met het beheren van softwarekosten, het verwerkingsvermogen verhogen, efficiëntie, flexibiliteit en nog veel meer.

Daarnaast dragen CPC's bij aan de betrouwbaarheid en fouttolerantie van mainframesystemen door redundantie en failover-mogelijkheden te bieden. Over het geheel genomen spelen CPC's een cruciale rol bij het optimaliseren van de prestaties en bestendigheid van mainframe-omgevingen en zorgen ze ervoor dat deze kunnen voldoen aan de eisen van moderne zakelijke toepassingen.

De configuratie van de mainframe omgeving varieert afhankelijk van de manier waarop deze door IBM wordt opgezet en de specifieke behoeften van de organisatie. Dit betekent dat de configuratie kan verschillen wat betreft de centrale processor en de System Assistance Processors en andere componenten dat op de mainframe computer kan worden aangesloten.

De configuratie kan bijvoorbeeld zijn dat er een enkele centrale processor is met twee System Assistance Processors, of meerdere General Processing units met meerdere System Assistance Processors. De configuratie van je omgeving hangt uiteindelijk af van wat de behoeften zijn en afhankelijk van deze behoeften wordt een overeenkomst gesloten met de leverancier van het systeem. Deze configuratie wordt uitgevoerd door IBM in het kader van het overeengekomen contract.

Specialty Processors (SAP)

Binnen een mainframe computer zijn er verschillende soorten processors waarvan specialty processor die ook wel bekend zijn als System Assistance Processor (SAP) en general purpose processor of centrale processor, deel uitmaken. Een general purpose processor kan worden vergeleken met de centrale processor unit van de hedendaagse moderne computers. Specialty processors zijn daarentegen aanvullende processen die draaien naast de general purpose processor.

Het fundamentele concept achter specialty processors is de mogelijkheid die het biedt om de prestaties van processors voor algemene doeleinden (General purpose processor) te verbeteren. Met andere woorden, het vergemakkelijkt de uitvoering van specifieke werklasten op specialty processors, waardoor niet alle taken op een enkele processor hoeven te worden uitgevoerd.

In de kern zijn beide soorten processors dezelfde processor. IBM, de leverancier van deze processors, configureert in overeenstemming met het overeengekomen contract de processors en levert deze aan de klant. Alle processors in de CPC beginnen als equivalente processorunits (PU's) die niet zijn gekarakteriseerd voor gebruik. Elke processor wordt door IBM gekarakteriseerd tijdens de installatie of op een later tijdstip. (Tram, 2010).

Karakteriseren betekent in deze context het beperken van het type code dat kan worden uitgevoerd op een bepaalde processor. (Redbooks, 2010).

Voordelen

Een van de meest aantrekkelijke eigenschappen van de System Assistance Processor is dat ze over het algemeen kost-effectiever in gebruik zijn. In tegenstelling tot de werklast die verwerkt worden door general purpose processors, zijn de werklasten die worden toegewezen aan deze System Assistance Processoren niet gebonden aan de licentiekosten van IBM of die van externe software leveranciers, wat resulteert in een aanzienlijke kostenbesparing.

Deze kostenbesparing is te wijten aan het feit dat de softwarecontracten van IBM onder andere gebaseerd zijn op het gebruik van de General Purpuse-processor. Voor een gegeven periode, bijvoorbeeld 1 maand, worden op basis van het maandelijkse maximum gebruik de kosten voor die maand berekend. Aangezien de Speciality-processors niet in dit contract zijn opgenomen, kan de noodzakelijke werkbelasting hierop worden afgehandeld zonder dat deze wordt toegevoegd. Het is vermeldenswaard dat deze processors wel een initiële aankoopkost hebben, maar in tegenstelling tot algemene processors hebben ze geen gebruikskosten.

De prijzen en licenties voor mainframes zijn complex en kunnen behoorlijk verwarrend zijn, maar in essentie is de maandelijkse factuur voor mainframe software van je organisatie gebaseerd op het gemiddelde piekgebruik gedurende de maand (C. S. Mullins, 2022).

Bovendien zijn deze processors veel goedkoper om aan te schaffen dan ”general purpose processors”. Daarnaast zijn SAP-processoren vooral interessant omdat:

Specialty processors kunnen worden aangeschaft voor een eenmalig bedrag per engine, inclusief gratis vervanging door snellere zIIP-engines bij het upgraden naar een nieuwe machine. Dit betekent opnieuw kostenbesparing (C. Mullins, 2023).

Naast de voordelen die hierboven zijn vermeld, hebben Specialtiy processoren geen beperking in de operationele processen. Dit staat ook wel bekend als Kneecapping of capacity setting. Dit maakt dat deze processors, zoals hieronder vermeld, hun volledige potentieel kunnen benutten en dat ze voordelig zijn wat betreft kosten en prestaties: IFL’s, SAP’s, zAAP’s en ICF’s werken altijd op de volledige snelheid van de processor omdat deze processoren ”niet meetellenïn de prijsberekeningen van software. (*Introduction to the New Mainframe: z/OS Basics*, g.d.)

Tot slot zijn System Assistance Processors ook gekend als Specialtiy Processor, zoals de naam al zegt, gemaakt voor specifieke werklasten. In de volgende sectie gaan we dieper in op hun functionaliteit.

Types

In de hedendaagse mainframe computers hebben we verschillende typen processors, in deze subsectie zullen we het hebben over de verschillende types System Assistance Processors en het kort hebben over hun functionaliteiten.

I. ICF

- ICF (Internal Coupling Facility) wordt gebruikt voor het verwerken van cyclussen van koppelingsfaciliteiten in een omgeving voor het delen van data.

Met ICF-processors kunnen meerdere LPAR’s met z/OS® gegevens beheren en werklast verdelen in een Parallel Sysplex® geclusterd systeem (IBM, 2023e).

Het is een belangrijk component van IBM mainframe systemen en zorgt voor belangrijke functies: het beheren van datasets, het verzekeren van de data-integriteit en het verbeteren van de systeemprestaties.

Een koppelingsfaciliteit is in feite een groot geheugenschrijfblok dat door meerdere systemen wordt gebruikt om het werk te coördineren. ICF's moeten worden toegewezen aan LPAR's die dan koppelingsfaciliteiten worden (IBM Corporation, g.d.-b).

II. IFL

- IFL (Integrated Facility for Linux) wordt gebruikt voor het verwerken van Linux on System Z werklast op IBM mainframe machines. Deze type SAP worden voornamelijk gekozen wegens de reden dat de software kosten voor het gebruik maken veel voordeliger zijn, voor het draaien van de besturingssystemen Linux en z/VM.

In de ibm redbook wordt het volgende vermeld over deze Assistant Processor: Een IFL is bijna precies hetzelfde als een normale centrale processor. Het enige verschil is dat de IFL twee instructies mist die de CP wel heeft en die alleen door z/OS worden gebruikt. Linux en z/VM gebruiken deze instructies niet (Redbooks, 2010).

III. Spare

Deze assistent-processor is een niet gecatogoriseerde processor en wordt geactiveerd wanneer er een storing optreedt in een van de processors. Dit geldt zowel voor de systeem-assistent-processor(en) als voor de algemene processor(en). Wanneer een van deze processors niet meer werkt, neemt de SPARE-processor het over. Zoals de naam ook impliceert, is het een soort reserve processor die in het geval van een fout de werklast overneemt, zoals hieronder wordt in de IBM documentatie vermeld (Corporation, g.d.-b).

IV. zAAP/zIIP

Over beide typen processors zijn 2 aparte subsecties gemaakt, aangezien we in het onderzoek dat we hier uitvoeren meer in detail kijken naar de specifieke functionaliteiten van de zAAP en zIIP processors en niet naar de andere typen processors.

De reden hiervoor is dat beide typen processoren een belangrijke rol spelen bij het uitvoeren van java-code. Deze processoren hebben de mogelijkheid om java code uit te voeren in tegenstelling tot de anderen. Om deze redenen is het nodig om deze processortypes meer in detail te bekijken.

2.2.6. zAAP

In het onderzoek dat hier wordt uitgevoerd dient er gesproken te worden over de IBM zEnterprise Application Assist Processor of afgekort zAAP. zAAP was in de eerste instantie de specialty processor die werd gebruikt voor het uitvoeren van onder

andere Java-code. Het werd ontwikkeld voor de executie van specifieke service georiënteerde architectuur zoals Java en XML.

Vanwege het feit dat zAAP de Specialty processor is voor het uitvoeren van Java code en het huidige onderzoek gaat over Java code performantie, is het nuttig om meer te weten te komen over zAAP.

Initieel is zAAP, net als andere speciality processoren, gemaakt om de werklast van de centrale processor of general processor te verminderen en ook om de softwarekosten te verlagen die door gebruik van de general processor ontstaan. Dit wordt ook door de ontwikkelaar IBM als een reden gegeven in de ibm documentatie waar aan hieronder aan wordt gerefereerd.

Dit wordt ook als motivatie gegeven door de ontwikkelaar IBM in de documentatie van IBM. zAAP specialty processoren zijn ontworpen om algemene rekencapaciteit vrij te maken en softwarekosten te verlagen voor bepaalde webgebaseerde en SOA-gebaseerde Db2 workloads, zoals Java en XML (Corporation, 2024).

Maar net als andere Assistant processors zijn er enkele functies uitgeschakeld, waaronder maar niet beperkt tot interrupt afhandeling. Hierdoor is het niet mogelijk is om een volledig besturingssysteem op deze specifieke processor te draaien ten opzichte van een andere SAP namelijk IFL waar dat het wel mogelijk is om bijvoorbeeld z/OS of Linux op te draaien. Deze soort processoren worden niet aangestuurd door een besturingssysteem waarop ze draaien, zoals de hedendaagse computers. zAAP's worden aangestuurd door z/OS dat op de general-purpose processor draait. In brede lijnen gebeurt de werking als volgt.
De mainframe computer waar op de besturingsysteem z/OS draait heeft kennis over de omgeving waar het draait, het heeft kennis over die processor en wanneer er behoefte is om bijvoorbeeld een JAVA programma uit te voeren kan het worden doorgestuurd naar die de processor, zodat zAAP de nodige instrucities kan uitvoeren.

Op een andere manier gezegd is het een processor die naast de algemene processor draait waar specifieke werklasten kunnen worden uitgevoerd die ook in de algemene processor kunnen worden uitgevoerd, hoewel het goedkoper is om deze in zAAP te draaien in plaats van in de algemene processor.

Tenslotte rond 2009, met de release van de z13 mainframe systemen, werd de zAAP processor niet meer gebruikt maar werden de bijbehorende functionaliteiten overgenomen door een gespecialiseerde processor genaamd zIIP (Lascu e.a., g.d.).

Dit betekent dat de System Assistant Processor met de naam zAAP niet meer ondersteund wordt sinds de uitgave van de Z13 Mainframe computers, maar dat er een andere processor met de naam zIIP is ontworpen die onder andere de werklast kan opvangen die zAAP had.

In de onderstaande sectie gaan we dieper in op dit ”nieuwe” zIIP processor.

2.2.7. zIIP

IBM z Integrated Information Processor, in het kort zIIP, is net als andere System Assistance Processors een Speciality processor die naast de general processor draait om de rekencapaciteiten te vergroten en de kosten te beheersen. Het werd voor het eerst gelanceerd met IBM system z9 (een zSeries mainframe computer).

Toen het voor het eerst werd uitgebracht was het doel ervan om de General processor te ondersteunen, meer specifiek voor DB2 werklasten. Sindsdien is deze processor ook nog in staat om andere werklasten te verwerken naast de originele DB2 werklasten. Het versterkt de rol van het mainframe als data knooppunt van de onderneming door directe toegang tot DB2 kosten efficiënter te maken en de behoefte aan meerdere kopieën van de gegevens te verminderen (Corporation, g.d.-b).

Zoals eerder vermeld in het gedeelte waarin we het over zAAP hadden, werd zAAP met de release van het IBM systeem z13 onderdeel van zIIP. Met andere woorden, de werklasten uitgevoerd door zAAP worden nu uitgevoerd op zIIP en zAAP wordt sindsdien niet langer ondersteund.

Dit is niet het enige dat zIIP als functionaliteit heeft, nadat IBM informatie onthulde over de achterliggende technologie van zIIP. Konden ook andere software leveranciers gebruik maken van deze reeks processoren.

Bovendien werden er ook functionaliteiten aangeboden die voor zAAP nog niet beschikbaar waren zoals het uitvoeren van Python code. Voorheen was het in zAAP niet eens mogelijk om Python werklasten uit te voeren.

Daarnaast heeft IBM met de opkomst van Artificial Intelligence het mogelijk gemaakt om AI-toepassingen in de mainframeomgeving toe te brengen. Het gebruik van deze processor biedt voordelen zoals de veerkracht van systemen verbeteren, digitale transformatie versnellen, overstappen op hybride cloud, Database en AI, BI-ERP- en CRM-toepassingen en andere.

Zoals eerder werd vermeld is z Integrated Information Processors (zIIP) een System Assistance Processor. Dit betekent dat het naast de centrale processor draait. Het gebruik van dit soort processors is interessant omdat het gebruik ervan niet in het contract is inbegrepen en dat maakt ze kosteneffectief. Maar vóór z16 was er een beperking van het aantal zIIP-processors dat je in een mainframecomputer kon krijgen. De verhouding was dat je voor elke centrale processor 2 zIIP's kon kopen. Met z16 is deze beperking niet meer nodig, wat betekent dat er een hoger aantal zIIP's kan worden aangeschaft dan voorheen. Dit is zoals hieronder vermeld met de introductie van Z16 systemen voortgezet als een van de selling points (Corporation, g.d.-a).

2.3. X86 Computer systeem

Wat er hier wordt bedoeld met X86 computer systemen is computers waarvan de centrale processor met de X86 instructies set gebaseerde processor architectuur. Waarom we deze gekozen hebben word in de hoofdstuk hier onder in detail besproken.

2.3.1. Waarom X86?

Nu kan de gevraag gesteld worden waarom X86 of 64 bit processoren. De keuze om 64bit procesoren te kiezen lag aan het feit dat 64bit gebaseerde systemen een van de betaalbare systemen zijn waar het ruim publiek tot aanmerking kan komen. Ook wegens de reden dat het een van de meest veelvoorkomen processors zijn die wordt gebruikt in persoonlijk computers als op server platform(DTC1, 2022).

Bovendien is de architectuur bekend om zijn hoge prestaties en compatibiliteit met een breed scala aan software, waardoor het een populaire keuze is voor pc's en laptops wat het een aantrekkelijke keuze maakt(Wind River, g.d.).

Tenslotte zijn er verschillende x86 systemen waarvan de prijzen variëren, dit is gebaseerd volgens hun specifieke doeleinden. In vergelijking met de initiële mainframe kosten is dit een veel goedkopere optie. Daarnaast hebben de x86 systemen geen consumptiekosten in tegenstelling tot de mainframe systemen.

2.3.2. Besturingsysteem

Er zijn heel wat soorten besturingssystemen die draaien op computers met een 64bit-processor. Vaak hebben deze systemen ook de mogelijkheid om te virtualiseren met behulp van een hypervisor. Bijvoorbeeld, op een windows of linux besturingsysteem is het mogelijk om een virtualisatie tool te gebruiken zoals vmware en virtual box voor windows en FVM voor fedora (linux).

Zoals vermeld door Lenovo is er een breed scala aan besturingssystemen op x86-processoren, waaronder Windows, Linux-distributies en vele andere (Lenovo, g.d.). In deze onderzoek zal de nadruk liggen op windows als besturingssysteem.

2.3.3. Processor

In deze hoofdstuk zal er een meer gedetaileerde uitleg plaats vinden over processoren met een X86 architectuur.

In de hoofdstuk “Waarom X86” werd eerder vermeld dat x86 een veelgebruikte computerarchitectuur voor centrale verwerkingseenheden (CPU’s) is. Het is de dominante architectuur geworden voor persoonlijke computers en servers. De naam ”x86” is afgeleid van de 8086, een vroege processor van Intel. x86 CPU’s maken gebruik van een complex instruction set computer (CISC) ontwerp, waardoor ze meerdere instructies in één cyclus kunnen uitvoeren(Lenovo, g.d.).

Deze x86 architectuur kan verward worden met de x86-64. De x86 is een processor architectuur terwijl dat de x86-64 een verwijzing is naar het aantal bits dat het potentieel heeft om gegevens te verwerken namelijk 64 bits per klokcyclus.

2.4. Algoritmen

De algoritmen die in dit onderzoek worden gebruikt, zullen worden getest op hun snelheid en op hun gebruik van werkgeheugen. Het is bedoeld om met deze algoritmen rekenintensieve werklasten te creëren, zodat een soort meetbare simulatie van werkzaamheden kan worden uitgevoerd tijdens het uitvoeren van deze algoritmen. Van deze werklasten zal voor elk het gebruikte werkgeheugen, de totale tijd nodig om de benodigde bewerkingen uit te voeren als de processortijd bijgehouden en vergeleken worden tussen de twee computersystemen.

2.4.1. Priemgetallen

Een priemgetal is volgens de Oxford English Dictionary van Oxford Languages “een getal dat niet gedeeld kan worden door een geheel getal (zonder rest) anders dan zichzelf en één. (Oxford University Press, 2024)

Priemgetallen staan bekend om hun belang in de fundamentele stelling van de rekenkunde. Er zijn een aantal verschillende algoritmen om priemgetallen te berekenen, waaronder twee algoritmen die in deze bachelorproef zijn uitgekozen en redelijk bekend zijn.

De reden waarom in dit onderzoek voor priemgetallen is gekozen, is omdat de gekozen algoritmen en methode ervoor kunnen zorgen dat er computationeel intensief kan worden gewerkt. Dit komt wegens de complexiteit van de achterliggende wiskunde en wegens de reden dat priemgetallen geen voorspelbare patronen hebben. Deze unieke eigenschap word als volgt beschreven in de paper waar dat 2 priem alogirtmen worden vergeleken. Priemgetallen zijn de unieke getallen omdat deze getallen niet kunnen worden gedeeld door andere positieve gehele getallen, behalve 1 en zichzelf(Abdullah e.a., 2018).

Zeef van Eratosthenes

De zeef van Eratosthenes is een algoritme dat al heel lang bestaat; het onderliggende idee is gebaseerd op het feit dat de veelvouden van een priemgetal zelf geen priemgetallen zijn. Bij het zoeken naar priemgetallen kunnen dus alle veelvouden van elk priemgetal worden doorgestreept. Dit elimineert veel getallen die anders zonder reden zouden worden getest en bespaart dus tijd (Johnson, 2023).

De werking van het algoritme, zoals beschreven in de paper van **Denis Xavier Charles**(Shaffer & Dickinson, 2002) van de Universiteit van Wisconsin-Madison, wordt als volgt toegelicht:

Voor het vinden van alle priemgetallen tot een bepaalde grens x. Neem een lijst van de getallen 2,3,— , |x|. Selecteer 2 als een priemgetal en streep alle veelvouden van 2 door. Omdat 3 in dit geval niet doorgestreept is, moet 3 een priemgetal zijn. Streep de veelvouden van 3 door omdat ze composiet zijn, en kies dan het volgende getal dat nog steeds niet doorgestreept is en herhaal. Als na een stap het volgende niet-gekruiste getal groter is dan $\sqrt{x}$ is, stop dan. In deze fase zijn alle getallen die niet zijn weggestreept priemgetallen (Shaffer & Dickinson, 2002).

Het enigste verschil tussen de Originele algoritme en het gebruikte is dat er een controle is voor dat het gecontroleerd getal geen negatief getal is. Dit omdat hier de maximale waarde voor int word genomen en in de p*p bewerking deze getal boven het maximale waarde gaat en het een negatief getal van maakt en de algoritme niet laat werken zoals het her verwacht word.

Tijds complexiteit: Laten we de complexiteit van dit algoritme analyseren. Het gebruikt 0(n) bits ruimte voor de array s. Initialisatie neemt O(n) operaties in beslag. De totale tijd die besteed wordt aan het vinden van het volgende priemgetal is gelijk aan het aantal keren dat we een priemgetal toevoegen aan p, wat maximaal $O(\sqrt{n})$ is. Tot slot is de tijd die besteed wordt aan het verwijderen van veelvouden hoogstens: $\sum_{p \leq \sqrt{n}} \frac{n}{p} = O(n \log \log n)$ (Jonathan Sorenson, 1990).

Geheugen complexiteit:
Het gebruikt 0(n) bits ruimte voor de array s. Initialisatie neemt O(n) operaties in beslag (Jonathan Sorenson, 1990).

De zeef van Eratostenes heeft O(n log log n) optellingen nodig met O(n) bits staraoge om alle priemgetallen tot n te vinden.

Algorithm 1 Zeef van Eratosthenes

```
1:  procedure ZeefVanEratosthenes(n)
2:      Initialiseert de booleaanse array voor priemgetalmarkering:prime[0 ... (n + 1)]
3:      for i ← 0 to n do
4:          prime[i] ← true
5:      end for
6:      for p ← 2 to √n do
7:          if prime[p] then
8:              for i ← p^2 to n step p do
9:                  if i ≥ 0 then
10:                     prime[i] ← false
11:                 end if
12:             end for
13:         end if
14:     end for
15: end procedure
```

Proef deling

Zeef van Eratosthenes

De proef deling of in het Engels trial division is een redelijke gemakkelijke maar trage manier om voor een willekeurig getal te bepalen of het een priemgetal is of niet aangezien het volledige getal wordt overlopen. Deze algoritme is wegens zijn werking slechter dan de Zeef van Eratosthenes. Wegens deze eigenschap is deze algoritme niet enkel bedoeld om de prestatie te vergelijken maar ook een baseline te verkrijgen tegen over de Zeef algoritme.
Er zijn verschillende soorten implementaties van het Proef deling-algoritme, maar in de basis is de werking als onderstaand beschreven.
Als input wordt verondersteld van een willekeurig getal n, de waarden voor deler d=2 en teller c=0 worden gedefinieerd en als volgt vindt de logica plaats. Indien d nog kleiner dan of gelijk aan n is, dan wordt er gecontroleerd of n modulo d geen rest heeft, wanneer de conditie vervult wordt dan stijgt c met een waarde van 1. Na deze controle stijgt de eerder vooraf gedefinieerde d ook met een waarde van 1. Buiten de while-lus waarbij d zal incrementeren met 1 tot het gelijk is aan n, vindt de laatste check plaats, als c gelijk is aan 1 dan wordt de output True en indien niet

dan wordt False als output geretourneerd (Samuel S. Wagstaff, 2005).

Algorithm 2 Pseudocode voor proef deling `td(n)`

```
1:  function TD(n)(Samuel S. Wagstaff, 2005)
2:      d ← 2
3:      c ← 0
4:      while d ≤ n do
5:          if n  mod d = 0 then
6:              c ← c + 1
7:              d ← d + 1
8:          end if
9:      end while
10:     if c = 1 then
11:         return True
12:     else
13:         return False
14:     end if
15: end function
```

De implementatie van de proef deling algoritme zal deels anders zijn dan bovenstaande basis algoritme. De werking van deze variatie van het algoritme is als volgt beschreven:

Algorithm 3 Pseudocode voor de functie`Proef deling(start, end, numParts)`

```
1:  function PROEF DELING(start, end, numParts)
2:      primes ← empty list
3:      rangeSize ← (end − start + 1)
4:      partSize ← rangeSize/numParts
5:      remainingNumbers ← rangeSize  mod numParts
6:      for i from 0 to numParts − 1 do
7:          partStart ← start + i × partSize + min(i, remainingNumbers)
8:          partEnd ← partStart + partSize − 1 + (i < remainingNumbers?1 : 0)
9:          for num from partStart to partEnd do
10:             if ISPRIME(num) then
11:                 add num to primes
12:             end if
13:         end for
14:     end for
15:     return primes
16: end function
```

De methode die zal worden gebruikt is een methode met 3 parameters namelijk start, end en numParts. Dit algoritme geeft als resultaat een lijst van priemgetallen binnen het opgegeven bereik.

De parameters **start** en **end** geven het bereik van waar dat de priemgetallen moeten gevonden worden. De derde parameter **numParts** die gebruikt wordt staat voor het aantal segmenten waarin het gegeven bereik wordt verdeeld voor parallelle verwerking. Dit is ook een onderdeel dat het onderscheid tussen de basis logica van de methode. Het zorgt er voor dat bij het vinden van priemgetallen de bereik kan opgedeeld worden in kleinere delen en deze delen gelijktijdig kan verwerkt worden door gebruik te maken van multi-threading of parallel executie.Het is wel belangrijk te melden dat de optimale deling afhankelijk is van het bereik waarvan de waardes uit moet berekent worden.

In het begin van deze methode wordt er een lege lijst geinitializeerd om de gevonden priemgetallen op te slaan. Vervolgens wordt er in **rangeSize** en **partSize** het bereik als de deel grote berekend. De variabele **remainingNumbers** staat voor het aantal resterende getal of getallen na het verdelen van het bereik in gelijke delen.

In de binnenlus worden de verdeelde delen doorlopen, de variabelen **partStart** en **partEnd** worden de begin als eind waardes voor dat deel berekend en toegewezen.

Tenslotte wordt er in de binnen lus waar dat de priemgetallen worden gedetecteerd met de methode **isPrime** en toegevoegd in de lijst.

Tijds complexiteit: Om de tijds complexiteit van deze algoritme te kunnen weergeven is het nodig om enkele aspecten eerst te bekijken.

De proef deling algoritme zal over elke getal één per één itereren in het gegeven bereik en bepalen of het gegeven getal een priemgetal is. De tijd dat nodig is om te itereren is evenredig met het aantal getallen in het bereik, die we hier beschouwen als n. Bij deze controle wordt voor efficiëntie een wortel gebruikt. Dit wordt gedaan om te kunnen voorkomen dat het algoritme onnodige controles zou moeten uitvoeren.

Als *n* deelbaar is door *d* dan is *n/d* ook een deler van *n* Als *d* groter is dan de wortel van *n* dan moet *n/d* kleiner zijn dan de wortel van *n* Daarom is het voldoende om alleen getallen tot aan de wortel te controleren(Samuel S. Wagstaff, 2005).

Door gebruikt te maken van bovenstaande eigenschap wordt de controle $\sqrt{n}$ keer uitgevoerd.

Omdat in deze algoritme het bereik verdeeld word in verschillende delen wordt de controle **(n/numParts)** keer uitgevoerd wat uiteindelijk naar het volgende tijds complexiteit heeft: $O(n \log \log n)$

kortom de tijds complexiteit van het algoritme is ongeveer **O((n/numParts) * √n)**, waarbij n de grootte van het bereik is. Dit betekent dat bij gebruik van dit algoritme de tijd complexiteit kan variëren afhankelijk van het aantal gebruikte delen.

Geheugen complexiteit:
Wegens dat we enkel gebruik maken van een beperkte aantal variabelen en dat deze variabelen een constante hoeveelheid geheugen vereisen ongeacht de grootte van de data die wat er verwerkt wordt.

Daarnaast groeit de lijst waar dat de priemgetallen naarmate er meerdere priemgetallen worden gevonden, maar aangezien de aantal priemgetallen veel kleiner is dan de grootte van de gegevens die verwerkt worden kan de geheugen complexiteit constant beschouwd worden in relatie tot de grootte van de invoer waarde.

Kortom de geheugen complexiteit van dit algoritme is **O(1)** omdat het geheugengebruik niet schaalt met de grootte van de invoer. Er wordt slechts een constante hoeveelheid geheugen gebruikt

2.4.2. Fibonacci

De fibonaccireeks is normaal ook een bekend wiskundig algoritme, waarbij voor een reeks getallen de waarde van het getal de som is van de vorige twee getallen, bijvoorbeeld 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...

In deze onderzoek gaan we echter gebruik maken van een recursieve implementatie van deze algoritme wegens de reden dat bij het gebruik van deze methode het algoritme veel meer resources en tijd in beslag neemt.
Deze implementatie biedt niet de meest efficiënte oplossing om Fibonacci-getallen te berekenen, vooral niet voor grote aantallen. Aangezien deze implementatie de Fibonacci-getallen recursief berekent zonder eerdere executies bij te houden. De inefficiëntie heeft echter ook gevolgen voor het gebruik van geheugen en processor tijd dat nodig is om de gewenste resultaat te calculeren. Zonder eerdere executies bij te houden berekent het algoritme dezelfde Fibonacci-getallen meerdere keren opnieuw, wat leidt tot onnodig CPU-gebruik. Bovendien creëren de recursieve aanroepen een groot aantal functieaanroepen op de call stack, wat tot meer werk geheugengebruik leidt. Als gevolg hiervan nemen zowel het gebruik van geheugen als de CPU-tijd merkbaar toe naarmate de grootte van de invoer toeneemt.

Algorithm 4 Recursive Fibonacci Algorithm

```
1: function FIBONACCI(n)
2:     if n ≤ 1 then
3:         return n
4:     else
5:         return FIBONACCI(n − 1) + FIBONACCI(n − 2)
6:     end if
7: end function
```

Tijds complexiteit: De tijds complexiteit van het recursieve Fibonacci algoritme zonder eerdere executies bij te houden is $O(2n)$. Dit komt door het feit dat voor elk Fibonaccigetal dat wordt uitgerekend, twee recursieve calls worden uitgevoerd.

Geheugen complexiteit: De geheugen complexiteit van het recursief Fibonacci algoritme zonder eerdere executies bij te houden is $O(2n)$. Dit is omdat de recursie stack lineair groeit met de waarde n van invoer, aangezien elke functie-aanroep geheugen in beslag neemt op de call stack tot dat de calculatie de gezochte waarde heeft bereikt.

Vanwege deze redenen dat hier boven worden vermeld is het voor dit uitgevoerde onderzoek een geschikt algoritme om de nodige berekeningen en vergelijkingen tussen de genoemde systemen uit te voeren.

3

Methodologie

In dit onderzoek tussen de prestaties van de Java Virtual Machine op een mainframe computer en een 64-bits computer met X86-architectuur worden computationeel intensieve algoritmen uitgevoerd en vervolgens beide systemen geanalyseerd.

Voor verduidelijking word er met computationeel intensieve algoritmen bedoeld dat de geschreven code berekeningen uitvoert met al bekende algoritmen die wiskundig intensief zijn. De reden waarom er reken intensieve algoritmen worden gebruikt om deze analyse te doen, is wegens de reden dat voor beide systemen een zelfde werk last wenst gesimuleerd te worden. Door gebruik te maken van een algoritme waar dat computationeel aspect het zelfde kan behouden worden is mogelijk om een meer representatieve vergelijking te doen op vlak van prestatie.

Deze testen werden opgedeeld in verschillende stappen:

3.1. Stappen

3.1.1. Stap 1: Onderzoek

Om te beginnen wordt er onderzoek uitgevoerd om informatie te verzamelen over de systemen die worden vergeleken met elkaar. Bovendien wordt er ook onderzoek verricht naar de programmeertaal als de wiskunde algoritmen die gebruikt zullen worden. De resultaten van deze onderzoek is terug te vinden in hoofdstuk 2.

3.1.2. Stap 2: Code schrijven

In deze fase lag de nadruk op het schrijven van Java-code met de bestudeerde algoritmen. Deze code word in eerste instantie zonder beperkingen getest op een

persoonlijke laptop.

3.1.3. Stap 3: Mainframe uitvoering

In deze stap wordt er voornamelijk onderzoek gedaan hoe dat Java code op de mainframe kan uitgevoerd worden. Bovendien zal er ook gekeken worden hoe dat de gebruikte systeem bronnen kunnen weergegeven worden op de mainframe omgeving.

3.1.4. Stap 4: Testen uitvoeren

In deze stap zijn de codes van stap 2 doorgevoerd. Deze codes zijn te vinden in sectie 5. Deze testen vinden plaats op beide systemen en de resultaten van deze testen worden bewaard in een afzonderlijk bestand. Tijdens deze stap kunnen ook bepaalde fouten en beperkingen worden opgemerkt, waardoor het nodig kan zijn om terug te keren naar stap 2 om de code aan te passen naar een verbeterdere versie.
Ook werden de resources zoveel mogelijk gelijk gehouden door onder andere de maximum gebruikbare heapsize vast te stellen met volgende commando : **java -Xmx1024m naam.java**

3.1.5. Stap 5: Analyse

De analyse wordt uitgevoerd door de verkregen cijfers uit de testen te vergelijken en hier vervolgens visualisaties over te maken. Aan de hand van deze visualisaties en de verzamelde cijfers wordt tot een conclusie gekomen.

3.2. Test omgevingen

De testen werden op 2 verschillende computer systemen uitgevoerd. Deze waren een mainframe computer gebaseerd op de Z-architectuur (z15) en op een laptop gebaseerd op de X86-64 architectuur.

De specificaties van de gebruikte mainframe omgeving is van confidentieel en werd door de onderneming gevraagd om deze eigenschappen niet publiek te maken. In het deel “Stand van zaken” wordt het systeem in het algemeen uitgelegd. Voor de hardware eigenschappen van de laptop is het als volgt:

Processor: AMD Ryzen 7 4800H with Radeon Graphics, basissnelheid 2900 MHz, 8 core(’s), 16 logische processor(s), Turbo Speed: 4.2 GHz
Geïnstalleerd fysiek geheugen (RAM): DDR4 16,0 GB /3200MHz

Naam van besturingssysteem: Microsoft Windows 10 Home
Besuringssysteem Versie: 10.0.19045 Build 19045

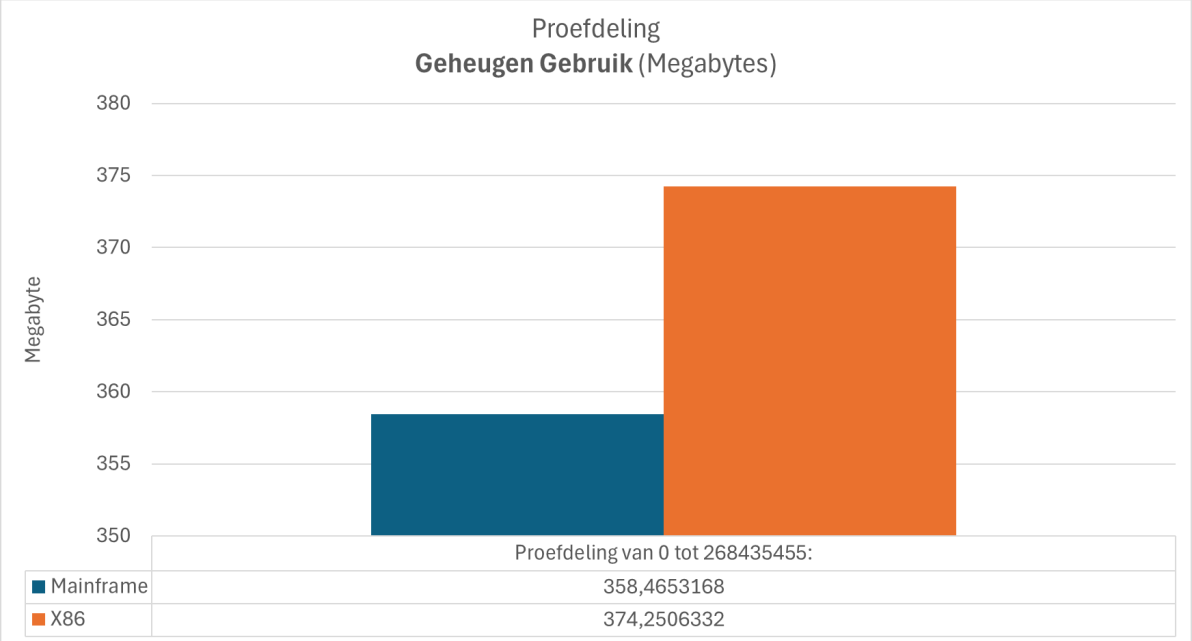
Tijdens de testen op de laptop die werden uitgevoerd zijn alle overbodige applicaties uitgeschakeld en de werd runtime envirement aangepast zodat beide systemen dezelfde bronnen in beschikking hebben om de behaalde resultaten zo eerlijk mogelijk met elkaar te zetten.

Ook waren de resultaten van beide systemen de gebruikte systeembronnen op dezelfde manier geëxtraheerd als een zelfde 1024mB heapsize ingesteld.

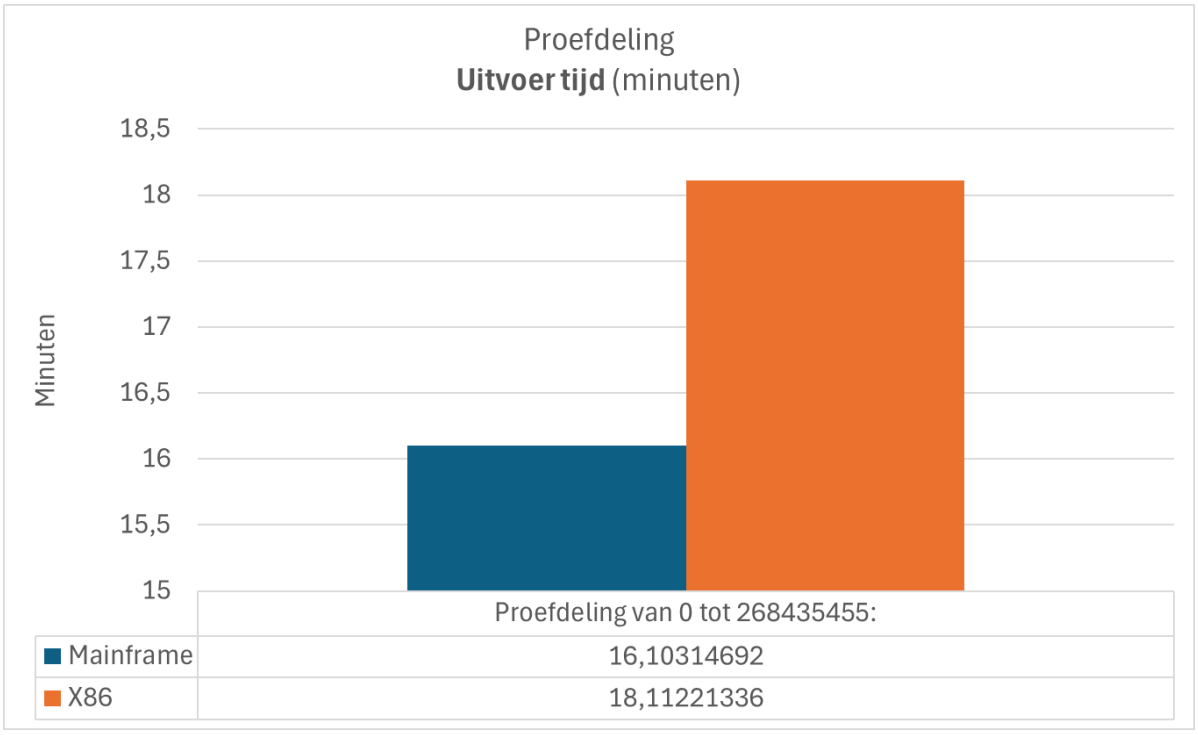
4

Resultaten

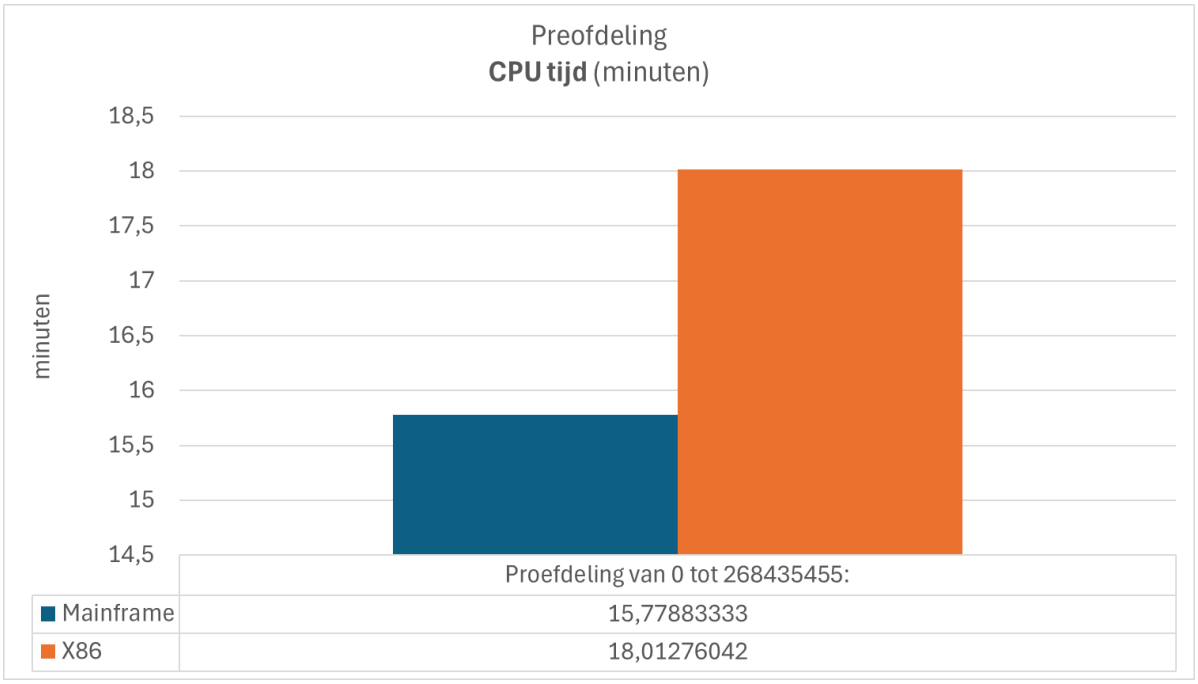
4.1. Proef deling



Figuur (4.1)
Grafische weergave van de hoeveelheid geheugen die nodig was om het algoritme uit te voeren, uitgedrukt in megabytes.

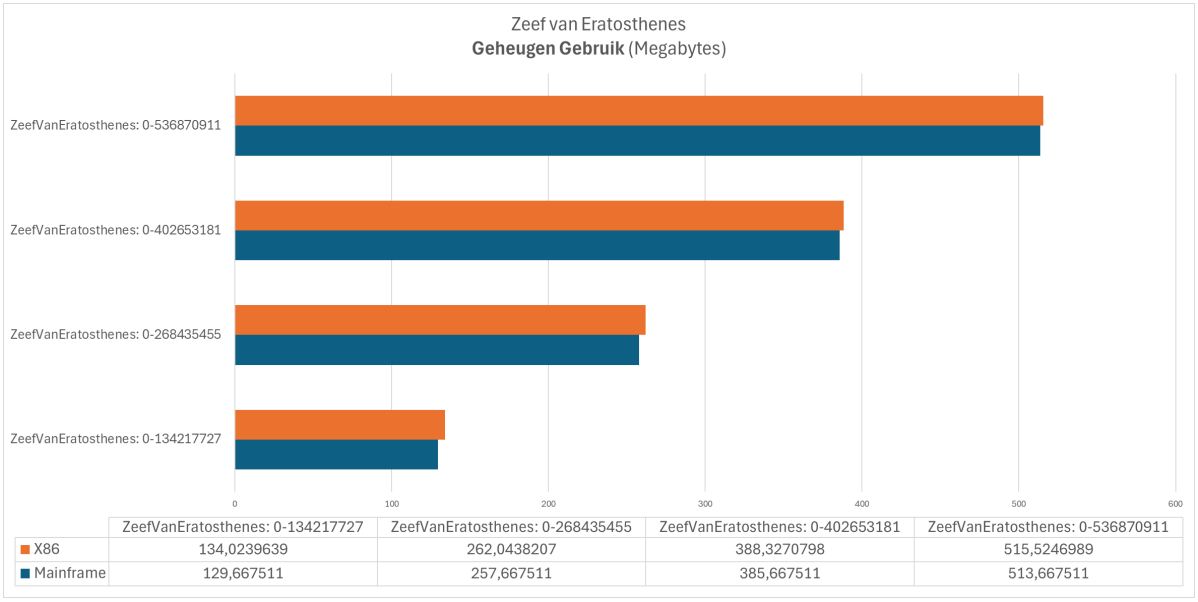


Figuur (4.2)
Grafische weergave van de totale tijd die nodig was om het algoritme uit te voeren, uitgedrukt in minuten.



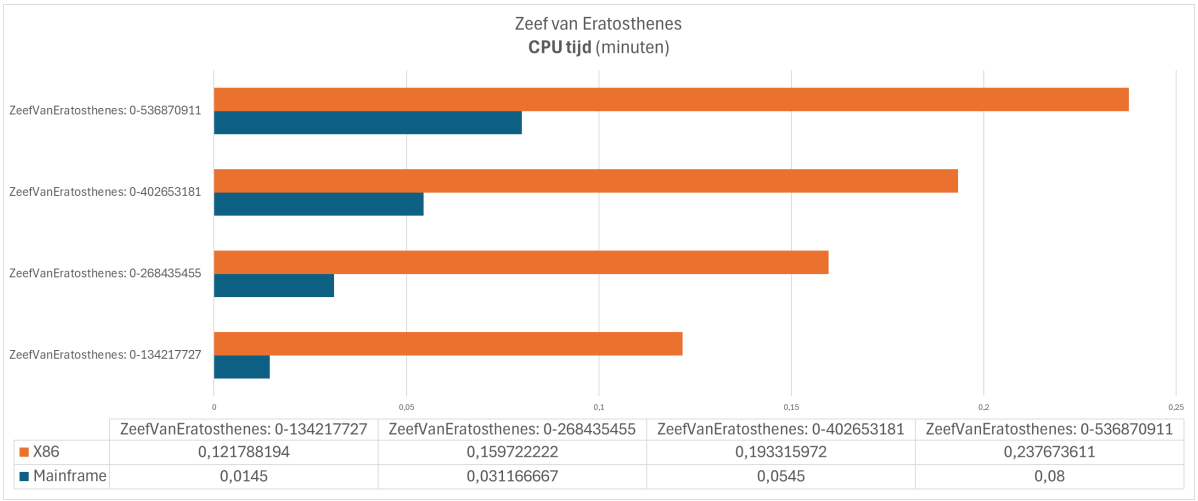
Figuur (4.3)
Grafische weergave van de processortijd die nodig was om het algoritme uit te voeren, uitgedrukt in minuten.

4.2. Zeef van Eratosthenes



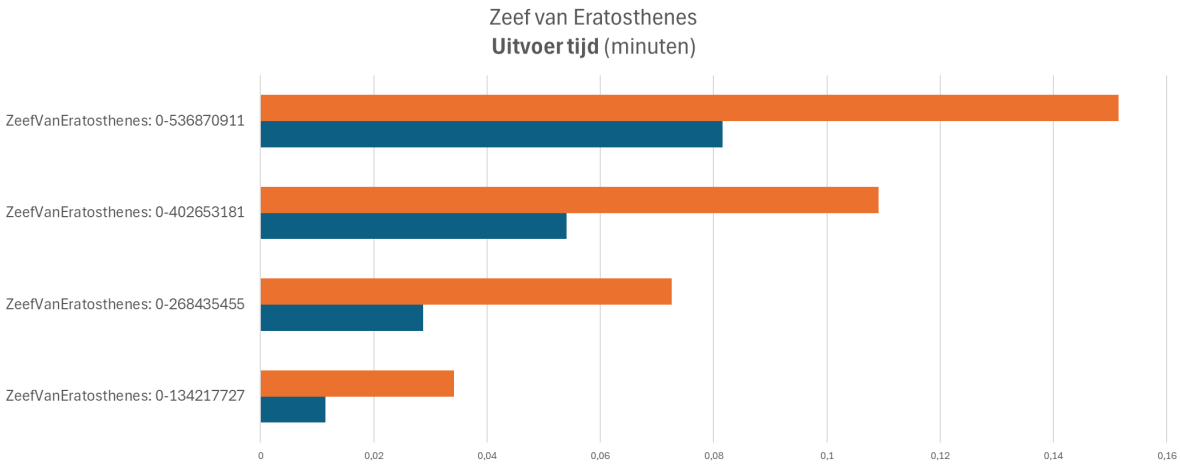
	ZeefVanEratosthenes: 0-134217727	ZeefVanEratosthenes: 0-268435455	ZeefVanEratosthenes: 0-402653181	ZeefVanEratosthenes: 0-536870911
X86	134,0239639	262,0438207	388,3270798	515,5246989
Mainframe	129,667511	257,667511	385,667511	513,667511

Figuur (4.4)
Grafische weergave van de hoeveelheid geheugen die nodig was om het algoritme uit te voeren, uitgedrukt in megabytes.



	ZeefVanEratosthenes: 0-134217727	ZeefVanEratosthenes: 0-268435455	ZeefVanEratosthenes: 0-402653181	ZeefVanEratosthenes: 0-536870911
X86	0,121788194	0,159722222	0,193315972	0,237673611
Mainframe	0,0145	0,031166667	0,0545	0,08

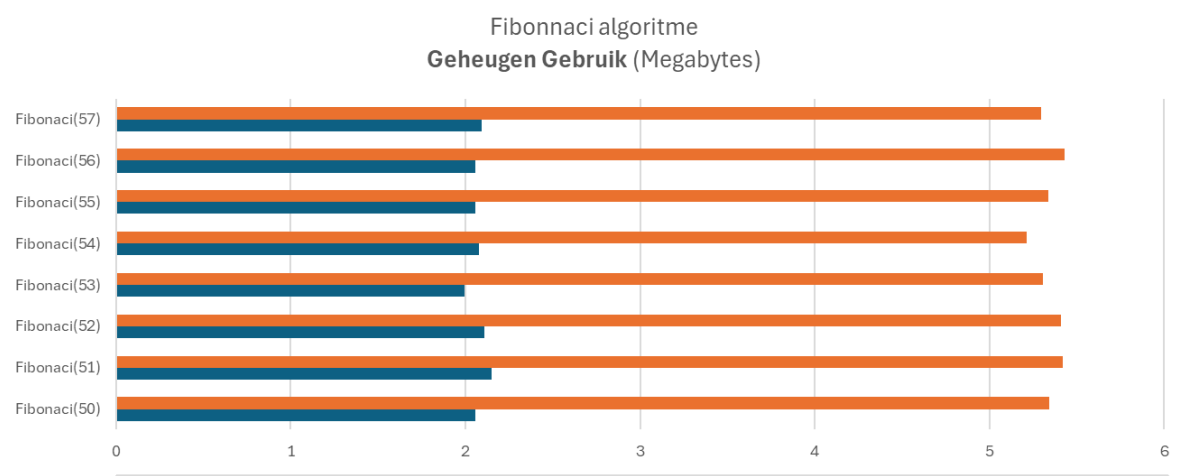
Figuur (4.5)
Grafische weergave van de processortijd die nodig was om het algoritme uit te voeren, uitgedrukt in minuten.



	ZeefVanEratosthenes: 0-134217727	ZeefVanEratosthenes: 0-268435455	ZeefVanEratosthenes: 0-402653181	ZeefVanEratosthenes: 0-536870911
X86	0,034205336	0,072582362	0,109142207	0,151516882
Mainframe	0,011449159	0,028670371	0,054013731	0,081625979

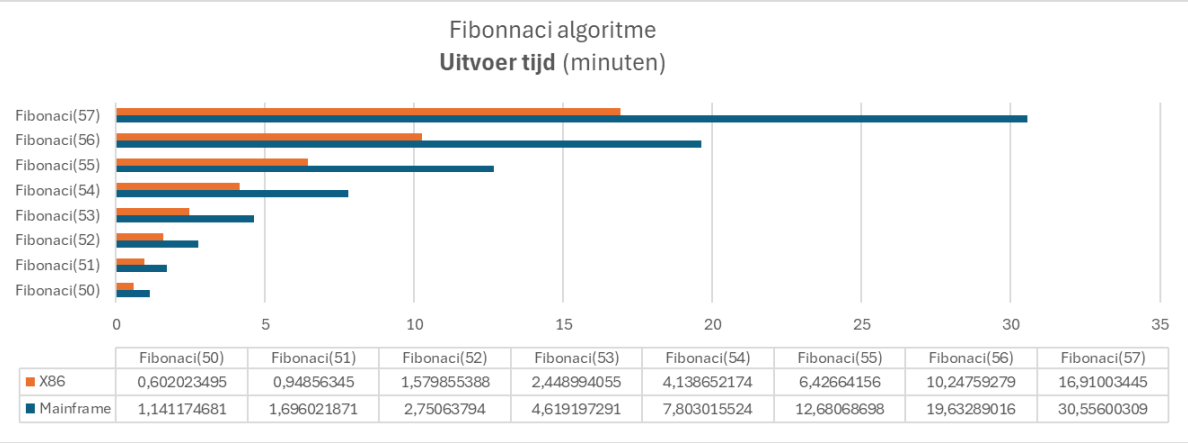
Figuur (4.6)
Grafische weergave van de totale tijd die nodig was om het algoritme uit te voeren, uitgedrukt in minuten.

4.3. Fibonaci



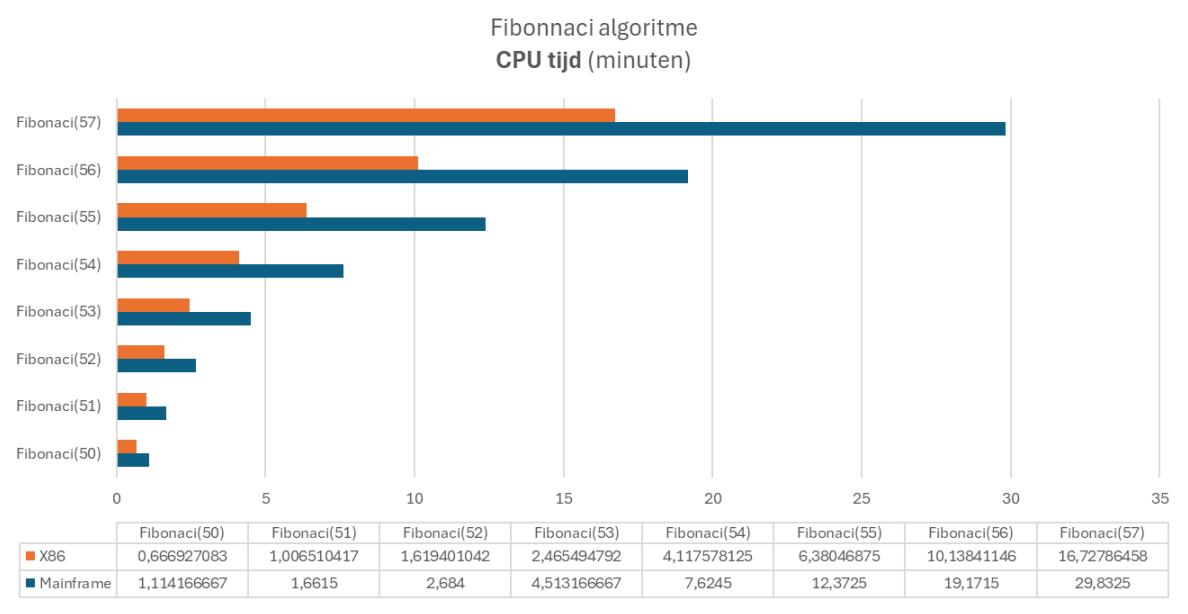
	Fibonaci(50)	Fibonaci(51)	Fibonaci(52)	Fibonaci(53)	Fibonaci(54)	Fibonaci(55)	Fibonaci(56)	Fibonaci(57)
X86	5,342111588	5,422021866	5,410562515	5,306499481	5,214443207	5,338636398	5,432100296	5,297334671
Mainframe	2,058914185	2,149841309	2,109825134	1,994438171	2,076622009	2,058914185	2,05632782	2,091957092

Figuur (4.7)
Grafische weergave van de hoeveelheid geheugen die nodig was om het algoritme uit te voeren, uitgedrukt in megabytes.



	Fibonaci(50)	Fibonaci(51)	Fibonaci(52)	Fibonaci(53)	Fibonaci(54)	Fibonaci(55)	Fibonaci(56)	Fibonaci(57)
X86	0,602023495	0,94856345	1,579855388	2,448994055	4,138652174	6,42664156	10,24759279	16,91003445
Mainframe	1,141174681	1,696021871	2,75063794	4,619197291	7,803015524	12,68068698	19,63289016	30,55600309

Figuur (4.8)
Grafische weergave van de totale tijd die nodig was om het algoritme uit te voeren, uitgedrukt in minuten.



	Fibonaci(50)	Fibonaci(51)	Fibonaci(52)	Fibonaci(53)	Fibonaci(54)	Fibonaci(55)	Fibonaci(56)	Fibonaci(57)
X86	0,666927083	1,006510417	1,619401042	2,465494792	4,117578125	6,38046875	10,13841146	16,72786458
Mainframe	1,114166667	1,6615	2,684	4,513166667	7,6245	12,3725	19,1715	29,8325

Figuur (4.9)
Grafische weergave van de processortijd die nodig was om het algoritme uit te voeren, uitgedrukt in minuten.

5

Code

5.1. Zeef van Eratosthenes

```java
package test;

import java.lang.management.ManagementFactory;
import java.lang.management.OperatingSystemMXBean;
import java.lang.management.MemoryMXBean;

public class PerformanceSieveOfEratosthenes {

    public static void main(String[] args) {
        long startTime = System.nanoTime();

        // Perform a computation-intensive task
        performTask();

        long endTime = System.nanoTime();

        // Measure elapsed time
        long elapsedTime = endTime - startTime;

        // Measure CPU time
        long cpuTime = getCpuTime();

        // Measure memory usage
        long memoryUsage = getMemoryUsage();
        System.out.println("");
        System.out.println("Elapsed Time: " + elapsedTime + " nanoseconds"
    );
        System.out.println("CPU Time: " + cpuTime + " nanoseconds");
```

```
28          System.out.println("Memory Usage: " + memoryUsage + " bytes");
29      }
30      private static void performTask() {
31          // SieveOfEratosthenes
32          int n=(Integer.MAX_VALUE/4*3 );
33          System.out.println(n);
34
35          // Initialize the boolean array for prime number marking
36          boolean[] prime = new boolean[ (n + 1)];
37          // Assume all numbers are prime initially
38          for (int i = 0; i <= n; i++)
39              prime[i] = true;
40
41          // Perform the Sieve of Eratosthenes
42          for (int p = 2; p*p>0 && p * p <= n; p++) {
43              if (prime[p]) {
44                  for (int i = p * p; i <= n; i += p) {
45                      if(i<0) {
46                          break ;
47                      }
48                      prime[i] = false;
49                  }
50              }
51          }
52
53          // Printing all prime numbers
54          //for (int i = 2; i <= n; i++)
55          //    if (prime[i]) {
56          //        System.out.print(i + " ");
57          //    }
58      }
59      private static long getCpuTime() {
60          OperatingSystemMXBean osBean = ManagementFactory.
    getOperatingSystemMXBean();
61          if (osBean instanceof com.sun.management.OperatingSystemMXBean) {
62              return ((com.sun.management.OperatingSystemMXBean) osBean).
    getProcessCpuTime();
63          } else {
64              return 0L;
65          }
66      }
67
68      private static long getMemoryUsage() {
69          MemoryMXBean memoryBean = ManagementFactory.getMemoryMXBean();
70          return memoryBean.getHeapMemoryUsage().getUsed();
71      }
72
```

```
}
```

5.2. Proef deling

5.3. Fibonaci

```
package test;

import java.lang.management.ManagementFactory;
import java.lang.management.OperatingSystemMXBean;
import java.lang.management.MemoryMXBean;

public class PerformanceFibonaci {

    public static void main(String[] args) {
        long startTime = System.nanoTime();

        // Perform a computation-intensive task
        performTask();

        long endTime = System.nanoTime();

        // Measure elapsed time
        long elapsedTime = endTime - startTime;

        // Measure CPU time
        long cpuTime = getCpuTime();

        // Measure memory usage
        long memoryUsage = getMemoryUsage();
        System.out.println("");
        System.out.println("Elapsed Time: " + elapsedTime + " nanoseconds"
    );
        System.out.println("CPU Time: " + cpuTime + " nanoseconds");
        System.out.println("Memory Usage: " + memoryUsage + " bytes");
    }
    private static void performTask() {
        // Fibonacci
        int n = 50;
        long result = fibonacci(n);
        System.out.println("Fibonacci(" + n + ") = " + result);
    }
    public static long fibonacci(int n) {
        if (n <= 1) {
            return n;
        } else {
```

```
40              return fibonacci(n - 1) + fibonacci(n - 2);
41          }
42      }
43
44      private static long getCpuTime() {
45          OperatingSystemMXBean osBean = ManagementFactory.
    getOperatingSystemMXBean();
46          if (osBean instanceof com.sun.management.OperatingSystemMXBean) {
47              return ((com.sun.management.OperatingSystemMXBean) osBean).
    getProcessCpuTime();
48          } else {
49              return 0L;
50          }
51      }
52
53      private static long getMemoryUsage() {
54          MemoryMXBean memoryBean = ManagementFactory.getMemoryMXBean();
55          return memoryBean.getHeapMemoryUsage().getUsed();
56      }
57
58  }
```

6

Conclusie

Uit de resultaten van de verrichte onderzoeken kan worden afgeleid dat de prestaties voor het berekenen van priemgetallen de mainframe beter presteert op tijd als geheugen. Maar voor de fibonaci algoritme is het ook zichtbaar dat de x86 systeem een kortere uitvoer tijd als CPU tijd hebben vergeleken met de mainframe maar waarvan op vlak van geheugen presteert mainframe veel beter.

De uitgevoerde testen zijn echter niet voldoende en er zouden veel meer testen moeten worden uitgevoerd. De systemen waarop de testen zijn uitgevoerd hebben niet dezelfde doeleinden en de testen zouden beter uitgevoerd moeten worden op systemen die zijn ontworpen voor dezelfde doeleinden. Het zou representatiever zijn geweest als de testen waren uitgevoerd op een X86-gebaseerde server dan op een persoonlijke laptop.

Ook moet worden vermeld dat niet alle testen uitvoerbaar waren vanwege enkele beperkingen van de testomgeving in mainframe. Een hiervan was de CPU-limit wat er voor zorgde dat programma's niet zo lang konden draaien. Er is geprobeerd geweest om voor memory heaps vast in te stellen tijdens uitvoer zodat de 2 systemen dezelfde heap hadden.
Met bovenstaande zaken in oog houdend kan geconcludeerd worden dat de uitgevoerde prestatie analyse niet 100% representatief is en hier uit een conclusie geven niet haalbaar is. Er kan uit de literatuurstudie wel vermeld worden dat beide systemen hun eigen plus als min punten hebben.

A

Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1. Inleiding

In de steeds veranderende wereld van de software-ontwikkeling blijft Java een belangrijke rol spelen als een van de meest gebruikte programmeertalen. Volgens een publicatie op Redmonk staat Java op de 3e plaats van de meest gebruikte programmeertalen (O'Grady, 2023). Dit toont aan dat java nog steeds een populaire programmeertaal is.
Het voornaamste doel van dit onderzoek is om de verschillen in prestaties tussen de twee systemen te ontdekken en daarmee aan te tonen welk systeem betere prestaties levert. Daarnaast willen we laten zien hoe elk platform zich gedraagt onder verschillende werklasten, met als extra doelstelling inzicht te verschaffen in prestatievariaties onder verschillende belastingen.

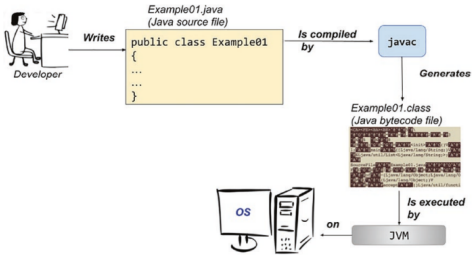
Deze vergelijkende studie belooft waardevolle inzichten op te leveren, vooral voor organisaties die mainframes inzetten voor hun softwareapplicaties, maar ook voor Java-ontwikkelaars.

A.2. Literatuurstudie

Java is wat we noemen een high-level programmeertaal waarmee een ontwikkelaar programma's kan schrijven die onafhankelijk zijn van een bepaald type computer. Talen op hoog niveau zijn gemakkelijker te lezen, te schrijven en te onderhouden (Cosmina, 2021b). Wat deze programeer taal zo aantrekkelijk maakt, is dat

het gebruik maakt van een zogenaamde Java Compiler.

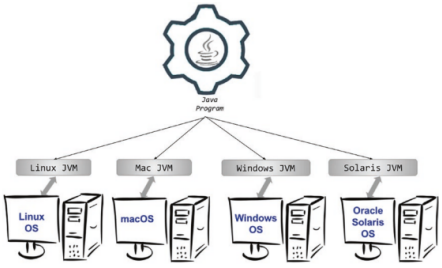
De Java-compiler is een programma dat een tekst kan lezen die is geschreven is in Java taal en het vertaalt in een bytecode die kan worden geïnterpreteerd door Java Virtual Machine die door een computer kan worden uitgevoerd (Samoylov, 2019).



Figuur (A.1)
Van Java-code naar machinecode

(Cosmina, 2021a)

De Java Virtual Machine (JVM) is een belangrijk onderdeel van dit ecosysteem, waardoor Java-toepassingen op verschillende hardware platforms kunnen draaien zonder dat ze aangepast hoeven te worden. Deze essentiële schakel functioneert als mediator tussen de geschreven Java-code en de fysieke machine waarop deze wordt uitgevoerd.



Figuur (A.2)
Een Java-programma uitvoeren op meerdere platforms

(Cosmina, 2021a)

Het onderzoek zal specifiek uitgevoerd worden op een Mainframe met een IBM virtual machine voor Java en een moderne pc. In de kern zijn mainframes krachtige computers met grote hoeveelheden geheugen en dataprocessors die miljarden eenvoudige berekeningen en transacties in realtime verwerken(I. IBM, g.d.).

In tegenstelling tot personal computers, die vooral voor persoonlijk gebruik zijn gemaakt zoals het naam zelf zegt, is het mainframe een commercieel gebruikte machine met aanzienlijk meer rekenkracht.

Onderzoek is uitgevoerd naar de werking van de IBM JVM, zoals gedocumenteerd in het boek ”Pro IBM® WebSphere® Application Server 7 Internals”waarin een gedetailleerde beschrijving van de IBM JVM wordt gegeven(12 & Watts, 2018).

Daarnaast is IBM zelf een van de supporters van de JVM met hun IBM JVM waar ze vanaf het uitvoeren tot het tuning voor de IBM JVM. Met dit geeft IBM aan hoe Java SE Runtime Environment configureren om de prestaties en het gebruik van systeembronnen af te stemmen en gelijkaardige diepgaande informatie (IBM, g.d.-a).

Apart van het voorgaande zijn er talloze onderzoeken gedaan naar het optimaliseren van het geheugenbeheer van Java.
zoals: Memory Management for Real-Time Java (Higuera e.a., 2004). Dit artikel behandelt het verbeteren van de prestaties van geheugenbeheer voor realtime Java-toepassingen.

Maar voor de rest is er geen direct vergelijkend onderzoek waarbij 2 verschillende platformen worden vergeleken. Meer specifiek wordt hier een x86 Windows systeem vergeleken met een mainframe waarbij de beschikbare middelen beperkt zijn voor een eerlijk vergelijkend onderzoek.

A.3. Methodologie

In de eerste fase ligt de nadruk op een literatuurstudie waarbij de volgende punten worden verduidelijkt:

1. Wat is Java & JVM
2. Evolutie van Java op het mainframe
3. Tuning voor de IBM virtual machine for Java

Er wordt verwacht dat de eerste faze 4 weken in beslag neemt.

De tweede fase focust op het opzetten van de verschillende test omgevingen. Hier wordt aandacht gegeven aan het beheren van bronnen op beide systemen om ervoor te zorgen dat de resultaten vergelijkbaar zijn.â
Na het voltooien van de vorige stappen zullen testen worden uitgevoerd om de prestaties van beide omgevingen te meten. Deze tests worden op basis van verschillende punten beoordeeld, namelijk:

- **Uitvoeringstijd van het algoritme**: De tijd die het algoritme nodig heeft om te worden uitgevoerd.

- **Resourcegebruik**: Het gebruik van rekenkracht (CPU) en geheugen tijdens de uitvoering.
- **Tijdscomplexiteiten**: Evaluatie van de tijdscomplexiteit van de algoritmen.

Er wordt verwacht dat de tweede faze 4 weken in beslag neemt.

In deze onderzoek zullen er priemgetallen gegenereerd worden door gebruik te maken van 2 algoritmes namelijk de Zeef van Eratostenes en de proefdeling. Het doel is om de werklast te verhogen door het aantal priemgetallen die gevonden moet worden te verhogen.
Er wordt gebruik gemaakt van priemgetallen, vanwege hun inherente complexiteit, vormen een uitdagende set van gegevens om te genereren(Abdullah e.a., 2018). Door deze getallen telkens te vergroten, wordt er gepland om een simulatie omgeving te creëren met een toenemende werklast voor de systemen. De resultaten van deze testen zullen worden vastgelegd en geanalyseerd. De verzamelde gegevens zullen worden gebruikt om grafieken op te stellen die de prestaties van beide systemen met de verschillende algoritmen visualiseren, waarbij specifiek wordt gelet op hoe ze omgaan met een grotere werklast in termen van het vinden van priemgetallen. Er wordt verwacht dat de laatste faze 3 weken in beslag neemt.

A.4. Verwacht resultaat, conclusie

Op het moment is het nog niet direct duidelijk welk systeem betere prestaties zal leveren. Er wordt verwacht dat de Mainframe betere prestaties zal hebben op het gebied van resourcegebruik in vergelijking met de Windows-computer, waarbij dezelfde hardware middelen worden toegewezen. Bovendien wordt er niet verwacht dat er over het algemeen aanzienlijke prestatieverschillen zullen zijn tussen beide platformen. Aan de hand van verschillende testen wordt er verwacht dat er een beslissend resultaat kan worden verkregen.

Bibliografie

12, D., & Watts, S. (2018). Java on the mainframe: Z/OS vs linux. *BMC Blogs*. https://www.bmc.com/blogs/java-mainframe-zos-linux/

Abdullah, D., Rahim, R., Apdilah, D., Efendi, S., Tulus, T., & Suwilo, S. (2018). Prime Numbers Comparison using Sieve of Eratosthenes and Sieve of Sundaram Algorithm. *Journal of Physics: Conference Series*, *978*(1), 0–1. https://doi.org/10.1088/1742-6596/978/1/012123

Byrne, J. C., & Cross, J. (2009). *Java for COBOL Programmers*. Charles River Media.

Corporation, I. (g.d.-a). *IBM Z Integrated Information Processor*. Verkregen maart 9, 2024, van https://www.ibm.com/products/z-integrated-information-processor

Corporation, I. (g.d.-b). *Mainframe Hardware: Processing Units*. Verkregen maart 1, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=concepts-mainframe-hardware-processing-units

Corporation, I. (2024, januari 16). *ZEnterprise Application Assist Processor (ZAAP) and DB2*. Verkregen maart 4, 2024, van https://www.ibm.com/docs/en/db2-for-zos/12?topic=performance-zenterprise-application-assist-processor-zaap-db2

Cosmina, I. (2021a). *Java 17 for Absolute Beginners: Learn the Fundamentals of Java Programming*. Apress. https://books.google.be/books?id=i1RYzgEACAAJ

Cosmina, I. (2021b). *Java 17 for Absolute Beginners: Learn the Fundamentals of Java Programming*. Apress.

Donohue, P. (2020). *Reducing the carbon footprint of computing*. International Business Machines Corporation. International Business Machines Corporation. https://www.ibm.com/downloads/cas/GYR3MWQN

DTC1. (2022, juli 14). *Guide for Server Processors*. Verkregen mei 3, 2024, van https://dtc1.com/guide-for-server-processors/

Higuera, T., Issarny, V., Banâtre, M., & Parain, F. (2004). Memory management for real-time Java: An efficient solution using hardware support*. *Real-Time Systems*, *26*(1), 63–87. https://doi.org/10.1023/b:time.0000009306.22263.59

Hunt, J. (2013). *Java and Object Orientation: An Introduction*. Springer London.

IBM. (g.d.-a). https://www.ibm.com/docs/en/was-zos/9.0.5?topic=jvm-tuning-virtual-machine-java

IBM. (g.d.-b). *IBM Linux Documentation*. Verkregen mei 25, 2024, van https://www.ibm.com/linux

IBM. (g.d.-c). *IBM z/Transaction Processing Facility Documentation*. Verkregen mei 25, 2024, van https://www.ibm.com/products/z-transaction-processing-facility

IBM. (g.d.-d). *Mainframe operating system: z/VSE*. Verkregen mei 25, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=systems-mainframe-operating-system-zvse

IBM. (g.d.-e). *z/OS Basic Skills Documentation: z/TPF*. Verkregen mei 25, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=systems-mainframe-operating-system-ztpf

IBM. (2023a). *Mainframe operating system: z/OS*. Verkregen mei 25, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=systems-mainframe-operating-system-zos

IBM. (2023b). *Mainframe operating system: z/VM*. Verkregen mei 25, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=systems-mainframe-operating-system-zvm

IBM. (2023c). *Mainframe Operating Systems (OS) | IBM*. Verkregen mei 25, 2024, van https://www.ibm.com/z/operating-systems

IBM. (2023d). *What are mainframe operating systems?* Verkregen mei 25, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=today-what-are-mainframe-operating-systems

IBM. (2023e). *z/VM 7.2 Documentation: Architectures and Specialty Processors*. Verkregen maart 4, 2024, van https://www.ibm.com/docs/en/zvm/7.2?topic=architectures-specialty-processors

IBM. (2023f). z/VM Software Virtualization | IBM. Verkregen mei 25, 2024, van https://www.ibm.com/products/zvm

IBM. (2024). https://www.ibm.com/docs/en/sdk-java-technology/8?topic=compiler-how-jit-optimizes-code

IBM, I. (g.d.). What is a mainframe? mainframe computing defined. *IBM*. https://www.ibm.com/topics/mainframe#:~:text=IBM%20mainframe%20computers%20are%20uniquely,advantage%20of%20unique%20hardware%20capabilities.

IBM Corporation. (g.d.-a). *IBM IT Infrastructure Demos*. Verkregen april 20, 2024, van %5Curl%7Bhttps://www.ibm.com/demos/it-infrastructure/index.html#C1315%7D

IBM Corporation. (g.d.-b). *Mainframe Hardware Processing Units*. Verkregen maart 8, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=concepts-mainframe-hardware-processing-units

IBM Corporation. (2010). *Network Availability: Mainframe*.

IBM Corporation. (2023). *Monitoring Advantages: Java*.

Introduction to the New Mainframe: z/OS Basics. (g.d.). IBM.

Johnson, R. (2023). The Sieve of Eratosthenes.

Jonathan Sorenson. (1990). *An Introduction to Prime Number Sieves* (Technical Report Nr. 909). University of Wisconsin-Madison. Verkregen mei 1, 2024, van https://research.cs.wisc.edu/techreports/1990/TR909.pdf

Krill, P. (2022, juli 16). *So why did they decide to call it Java?*

Lascu, O., Hoogerbrug, E., Leon, C. A. D., Palacio, E., Pinto, F., Sannerud, B., Soellig, M., Troy, J., & Yang, J. J. (g.d.). *IBM Z Functional Matrix: Architecture and Design Considerations*. Verkregen maart 4, 2024, van https://www.redbooks.ibm.com/redbooks/pdfs/sg248251.pdf

Lenovo. (g.d.). *What is x86?* Verkregen mei 3, 2024, van https://www.lenovo.com/us/en/glossary/x86/

Lindholm, T. (2013). *The Java Virtual Machine Specification: Java SE 7 edition*. Addison-Wesley.

Mullins, C. (2023). *ZIIPing Along with Mainframe Specialty Processors*.

Mullins, C. S. (2022, februari 15). *Understanding Mainframe Specialty Processors: zIIPs and More*. Verkregen maart 8, 2024, van https://cloudframe.com/understanding-mainframe-specialty-processors-ziips-and-more/

O'Grady, S. (2023). *The RedMonk Programming Language Rankings: January 2023*. https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/

Oracle Corporation. (2009). *Oracle Buys Sun*.

Oracle Corporation. (2024). *Duke - The Java Mascot*.

Oxford University Press. (2024). Prime Number.

Redbooks, I. (2010). *IBM Z Functional Matrix*. Verkregen maart 2, 2024, van https://www.redbooks.ibm.com/redbooks/pdfs/sg247748.pdf

Redbooks, I. (2020). *IBM Storage Solutions for IBM Blockchain Platform Version 2.5* [2021]. https://www.redbooks.ibm.com/redbooks/pdfs/sg248850.pdf

Samoylov, N. (2019). *Learn Java 12 Programming: A step-by-step guide to learning essential concepts in Java SE 10, 11, and 12*. Packt Publishing. https://books.google.be/books?id=-S6WDwAAQBAJ

Samuel S. Wagstaff, J. (2005). *Chapter 3: Introduction to Algorithms* (Technical Report). Purdue University. https://www.cs.purdue.edu/homes/ssw/chapter3.pdf

Shaffer, C. A., & Dickinson, C. (2002). *Implementing the Sieve of Eratosthenes* (tech. rap.). Department of Computer Science, University of Wisconsin-Madison. https://pages.cs.wisc.edu/~cdx/Sieve.pdf

Statistics & Data. (2023, september 21). *The Most Popular Programming Languages – 1965/2023*.

TIOBE Index for May 2024. (2024). https://www.tiobe.com/tiobe-index/

Tram, M. J. .-. H. S. D. (2010). *Mainframe Hardware Processing Units*. Verkregen april 21, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=concepts-mainframe-hardware-processing-units

University of Wollongong. (2024). *Understanding Operating Systems* (researchreport). University of Wollongong.

What is the Java Runtime Environment (JRE)? (2021). https://https://www.ibm.com/topics/jre

Wind River. (g.d.). *Leading Processor Architectures*. Verkregen mei 3, 2024, van https://www.windriver.com/solutions/learning/leading-processor-architectures#leading_processor_architectures

Winnie, D. (2021). *Essential Java for AP CompSci: From Programming to Computer Science*. Apress.