LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 146 – Abgabe zu Aufgabe A400
Sommersemester 2023

Gregorius Nico Toreh Rafael Buala Lahagu Raymond King Setia

1 Einleitung

Um prüfen zu können , ob eine Nachricht korrekt und fehlerfrei übertragen wurde, kommt die sogenannte Zyklische Redundanzprüfung (englisch : *cyclic redundancy check*, daher meist **CRC**) ins Spiel. Dieser Algorithmus, der 1961 von *W. Wesley Peterson* entwickelt wurde, erstellt den Prüfwert von Daten. Dadurch können Datenempfänger die Integrität der empfangenen Daten überprüfen und mögliche Fehler oder Korruptionen erkennen. Die CRC32-Berechnung basiert auf der Polynomdivision. Dabei wird der Bitstrom einer zu untersuchenden Nachricht der Länge l als Polynom mit binären Koeffizienten K_i dargestellt:

$$N = K_{l-1}K_{l-2}\dots K_1K_0 \quad \hat{=} \sum_{i=0}^{l-1} K_i \cdot x^i =: N(x)$$

Als Dividend für die Polynomdivision wird ein sogenanntes Generatorpolynom $G(x)$ verwendet. Der Rest der Polynomdivision von $N(x)/G(x)$ ist dann die zu berechnende Testsumme P. Formal gilt:

$$(CRC32(N, G) = P(x) \equiv N(x) \mod G(x))$$

Die Prüfsumme P wird einfach nach folgender Berechnung an die erste Nachricht N angehängt: $N' = (N \parallel P)$ .Nach dem Empfang der Nachricht N' führt der Empfänger die identische Polynomdivision unter Verwendung des identischen $G(x)$ durch. Ist das Ergebnis 0, kann mit hoher Wahrscheinlichkeit davon ausgegangen werden, dass die Nachricht korrekt übermittelt wurde.

In diesem Projekt implementieren wir eine Funktion **CRC32** mit der Signatur:

```
uint32_t CRC32(size_t len, const char msg[len], uint32_t generator)
```

Wir haben auch ein Szenario gezeigt, bei dem der CRC nicht funktioniert und die Schwachstelle des Programms aufzeigen könnte. Weitere Optimierungen wurden auch entwickelt, die die Effizienz des Prozesses verbessern können. Anschließend wird die Performanz jeder Implementierung gemessen und miteinander verglichen.

2 Lösungsansatz

Es gibt verschiedene Art and Weise, wie man der CRC32 Algorithmus implementieren kann. Wir werden uns in diesem Projekt der CRC32 Algorithmus auf 3 verschiedenen

Ansätzen ansehen : den algebraischen Ansatz, den bitorientierten Ansatz, und den tabelgesteuerten Ansatz.

Der algebraische Ansatz ist die mathematische Definition des CRC und besteht im Wesentlichen aus einer riesigen Polynomdivision. Dieser Ansatz enthält jedoch oft zu viel Mathematik, wenn man den Code schreiben möchte. Aus diesem Grund werden wir uns auch den bitorientierten Ansatz ansehen, bei dem die Polynomdivision in der Praxis direkt mit den Bits der Eingangsdaten durchgeführt wird.

Schließlich gibt es den tabellengesteuerten Ansatz, der dieselbe Arbeit auf schnellere und effizientere Weise erledigt, weshalb dies die Art und Weise ist, wie Programme in der realen Welt das CRC tatsächlich berechnen.[1]

2.1 Algebraischer Ansatz

Mehr oder weniger ist das CRC gemäß der Definition eine riesige Polynomdivision.Die Eingabe, deren Typ String ist, wird als Koeffizienten eines riesigen Polynoms interpretiert, das durch ein gegebenes CRC-Polynom, also das sogenannte Generatorpolynom G dividiert wird. Der Rest dieser Division ist dann die Prüfsumme. Im Folgenden möchten wir ein Beispiel betrachten, das den CRC4-Algorithmus mit einer 7 Bit langen Eingabe veranschaulicht. Anschließend werden wir erkennen können, wie eng verwandt der algebraische Ansatz und der bitorientierte Ansatz tatsächlich sind und dass eine herkömmliche Polynomdivision für den CRC-Algorithmus nicht geeignet ist.

Beispiel mit CRC4 :
Nachricht = 1 1 0 1 1 0 1
Generatorpolynom = 1 0 1 0 1 = $x^4 + x^2 + 1$
Der Divisor hat 5 Bits (daher ist dies ein CRC-4-Polynom), also werden 4 Nullbits an das Eingabemuster angehängt.
N mit angehängten Nullen = N + 0 = 1 1 0 1 1 0 1 0 0 0 0 = $x^{10} + x^9 + x^7 + x^6 + x^4$

$$
\begin{array}{r}
x^6 + x^5 \;\; - x^4 \qquad + x^2 - x + 1 \\
\hline
x^4 + x^2 + 1) \;\; x^{10} + x^9 \qquad + x^7 + x^6 \qquad + x^4 \\
\underline{- x^{10} \qquad - x^8 \qquad - x^6} \\
x^9 - x^8 + x^7 \\
\underline{- x^9 \qquad - x^7 \qquad - x^5} \\
- x^8 \qquad - x^5 \;\; + x^4 \\
\underline{x^8 \qquad + x^6 \qquad + x^4} \\
x^6 - x^5 + 2x^4 \\
\underline{- x^6 \qquad - x^4 \qquad - x^2} \\
- x^5 \;\; + x^4 \qquad - x^2 \\
\underline{x^5 \qquad + x^3 \qquad + x} \\
x^4 + x^3 \;\; - x^2 + x \\
\underline{- x^4 \qquad - x^2 \qquad - 1} \\
x^3 - 2x^2 + x - 1
\end{array}
$$

Abbildung 1: normale Polynomdivision, die wir kennen

Wie wir sehen können, enthält das Ergebnis der Polynomdivision, das die Prüfsumme für den CRC-Algorithmus darstellt, Koeffizienten, die größer als 1 und kleiner als 0 sind. Diese sind jedoch nicht mit Bits darstellbar, da Bits nur 1 darstellen können, wenn das Polynom existiert und 0, wenn es nicht existiert.

Was wir tun können, um diese in Bits darzustellen, ist die Verwendung eines mathematischen Tricks namens endlicher Körper. Algebra ist im Grunde der Satz von Regeln, mit denen man Symbole für Operationen manipulieren kann. Zum Beispiel :

$$\frac{x}{2} - 5 = 7$$

Wir addieren 5 zu beiden Seiten der Gleichung *(additive Eigentschaft der Gleichheit)*.

$$\frac{x}{2} - 5 + 5 = 7 + 5$$

$$\frac{x}{2} = 12$$

Wir multiplizieren beide Seiten der Gleichung mit 2 *(multiplikative Eigentschaft der Gleichheit)*.

$$\frac{x}{2} \cdot 2 = 12 \cdot 2$$

$$x = 24$$

Wenn wir also nach x auflösen, wenden wir jedes Mal, wenn wir Algebra machen, eine Reihe von Regeln an.Die Zahl, die wir verwenden, wenn wir Algebra anwenden, nennt man Körper.

In diesem Fall haben wir den Körper der reellen Zahlen $\mathbb{R}$ verwendet. Es gibt viele Körper in der Algebra, zum Beispiel die natürlichen Zahlen $\mathbb{N}$ und die komplexen Zahlen $\mathbb{C}$.Wir interessieren uns für endliche Körper. Wie der Name schon sagt, verwenden endliche Körper eine endliche Anzahl von Zahlen, aber die Regeln der Algebra funktionieren trotzdem gleich.

Ein Körper ist im Grunde genommen eine Menge von Zahlen. Um zu definieren, wie Operationen zwischen diesen Zahlen funktionieren, müssen wir Körperaxiome einführen. Unser Körper $\mathbb{F}_2$ besteht nur 2 Zahlen: 0 und 1.Also $\mathbb{F}_2 = [0, 1]$.

Assoziativität	$: a + (b + c) = (a + b) + c$	$\mid a \cdot (b \cdot c) = (a \cdot b) \cdot c$
Kommutativität	$: a + b = b + a$	$\mid a \cdot b = b \cdot a$
Identität	$: a + 0 = a$	$\mid a \cdot 1 = a$
Additive Inverse	$: a + (-a) = 0$	
Multiplikative Inverse	$: a \cdot \frac{1}{a} = 1$	
Distributivität	$: a \cdot (b + c) = (a \cdot b) + (a \cdot c)$	

Abbildung 2: Körperaxiome

Mit den oben erstellten Feldaxiomen können wir Addition, Subtraktion, Multiplikation und Division wie folgt definieren :

$0+0=0$	$0-0=0$	$0\cdot 0=0$	$\frac{0}{1}=0$
$0+1=1$	$0-1=1$	$0\cdot 1=0$	$\frac{1}{1}=1$
$1+0=1$	$1-0=1$	$1\cdot 0=0$	$\frac{1}{0}=\infty$
$1+1=0$	$1-1=0$	$1\cdot 1=1$	$\frac{0}{0}=$ *undefiniert*

Abbildung 3: neu definierte Operationen nach den Axiomen

Durch die Definition der Operationen auf diese Weise erfüllen wir alle Körperaxiome. Daher handelt es sich um einen gültigen Körper, obwohl es einige gegensätzliche Dinge gibt, die wir möglicherweise nicht erwarten.Zum Beispiel müssen wir das **additive Inverse** haben.Daher können wir sagen,dass $a+(-a)=0$ ist. In unserem Körper ist das additive Inverse einer Zahl die Zahl selbst. Zum Beispiel ist das additive Inverse von 1 ebenfalls 1, und auch $1+1=0$. Als nächstes betrachten wir ein Beispiel, in dem die Axiome angewendet wurden.

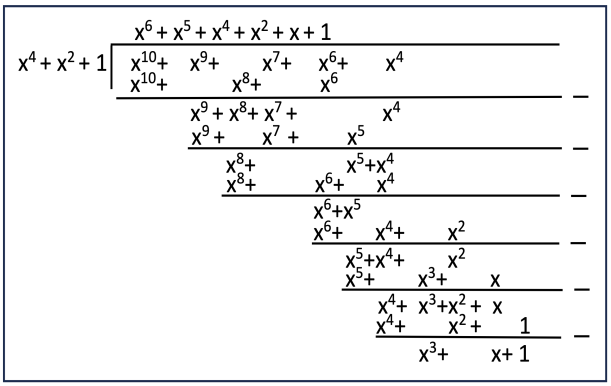

Abbildung 4: Polynomdivision mit vordefinierten Axiomen

Der Rest der Polynomdivision x^3+x+1 stellt die Prüfsumme 1011 für den CRC-Algorithmus dar. Jetzt kann man sehen, dass es keine Koeffizienten gibt, die größer als 1 und kleiner als 0 sind, und kann daraus schließen, dass dieses Polynom nun mit Bits darstellbar ist.

2.2 Bitorientierter Ansatz

Nun wird die dort oben definierten Operationen betrachtet, die auf den Körperaxiomen beruhen. Das Ergebnis der möglichen Eingabekombinationen für Addition und Subtraktion stellt tatsächlich die Wahrheitstabellen-XOR-Operation dar.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Abbildung 5: Wahrheitstabelle der XOR-Operation

Wir haben die Funktion

```
uint32_t CRC32_V2(size_t len, const char msg[len], uint32_t generator)
```

mit der Programmiersprache C implementiert. Anschließend können wir sehen, wie sich die XOR-Operationen verhalten und wie wir sie für die CRC-Berechnung ausnutzen können.

```
1  uint32_t CRC32_V2(size_t len, const char msg[len], uint32_t generator) {
2      uint32_t result = 0;
3      for (size_t i = 0; i < len; i++) {
4          result ^= (uint32_t)(msg[i] << 24);
5          for (size_t j = 0; j < 8; j++) {
6              if ((result & 0x80000000)) {
7                  result = (uint32_t)((result<<1)^generator);
8              } else{
9                  result <<=1;
10             }
11         }
12     }
13     return result;
14 }
```

Die Methode **CRC32_V2** nimmt die Eingabe in Form eines Nachrichtenarrays **msg** an und verarbeitet sie byteweise. Die Berechnung der CRC-Prüfsumme erfolgt jedoch bitweise innerhalb jedes Bytes. Dabei wird das Ergebnis schrittweise aktualisiert, indem die XOR-Operation auf die entsprechenden Bits angewendet wird. Am Ende wird die vollständige CRC-Prüfsumme zurückgegeben.

2.3 Ansatz mit Umsetzungstabelle

Die Art und Weise, wie dieser tabellengesteuerte Ansatz funktioniert, unterscheidet sich nicht wesentlich vom bitorientierten Ansatz. Aber anstatt eine XOR-Operation an der Bitfolge durchzuführen, erstellen und berechnen wir zunächst eine Umsetzungstabelle *(englisch : Lookup-Table)* mit dem eingegebenen Generatorpolynom.

Die Größe der Tabelle beträgt 256, da die Größe eines Characters *(char)* 8 Bit beträgt und es daher 256 mögliche Eingaben für das Character gibt. Nun werfen wir einen Blick auf die Methode, mit der man die Umsetzungstabelle erstellen kann.

```
void crc32TableFillSIMD(uint32_t* table, uint32_t generator) {
    __m128i genpoll = _mm_set1_epi32(generator);
    for (int i = 0; i < 256; i += 4) {
        __m128i res = _mm_slli_epi32(_mm_set_epi32(i+3, i+2, i+1, i), 24);
        for (uint8_t j = 8; j > 0; j--) {
            __m128i msb = _mm_srai_epi32(res, 31);
            res = _mm_slli_epi32(res, 1);

            __m128i r = _mm_and_si128(msb, genpoll);
            res = _mm_xor_si128(res, r);
        }
        _mm_storeu_si128((__m128i*)&table[i], res);
    }
}
```

Der Code verwendet SIMD-Anweisungen, um eine CRC32-Lookup-Tabelle mit dem gegebenen Generatorpolynom zu füllen. Die Funktion **crc32TableFillSIMD** nimmt einen Zeiger auf die Lookup-Tabelle **table** und das Generatorpolynom **generator** entgegen.

Der Code verwendet SIMD-Register, um mehrere Operationen parallel auszuführen und die Effizienz zu verbessern.Die äußere Schleife iteriert über die Indizes der Lookup-Tabelle. In jedem Schleifendurchlauf werden vier Einträge gleichzeitig berechnet.

Die innere Schleife führt acht Iterationen aus, um die CRC-Berechnung für jeden Index zu vollziehen. Am Ende wird das Ergebnis aus den SIMD-Registern in die Lookup-Tabelle geschrieben.

Man kann dann den Wert (0x00 -0xFF) aus der Lookup-Tabelle abrufen, anstatt XOR-Operationen durchzuführen. Aus diesem Grund müssen wir denselben Wert nicht immer wieder neu berechnen, wenn die Eingabe viele gleiche Werte enthält.

3 Korrektheit

Die Korrektheit des CRC32-Algorithmus ist entscheidend für die Gewährleistung der Datenintegrität. Dabei stehen zwei Hauptaspekte im Fokus: die deterministische Erzeugung des Prüfwerts und die zuverlässige Erkennung von Fehlern. Ein guter CRC32-Algorithmus erzeugt für eine gegebene Datenfolge immer denselben Prüfwert und hat eine hohe Wahrscheinlichkeit, fehlerhafte Daten zu identifizieren, während die Fehlalarmrate für korrekte Daten gering sein sollte.

3.1 Generatorpolynom

Beginnen wir damit, zu untersuchen, wie man ein geeignetes Generatorpolynom G auswählen kann, um die Wahrscheinlichkeit einer zuverlässigen Fehlererkennung zu maximieren[2]. Um dies zu veranschaulichen, betrachten wir ein Beispiel.

Der Empfänger bekommt ein Generatorpolynom $G = 1\,1\,0\,1\,0\,1$ und eine Nachricht N'.

Fallunterscheidung :
Fall 1 : $N' = 1\,\mathbf{1}\,0\,1\,1\,0\,0\,1\,0\,1 = x^9 + x^8 + x^6 + x^5 + x^2 + 1 \; mod\, G = 0$
Fall 2 : $N'_{Fehler} = 1\,\mathbf{0}\,0\,1\,1\,0\,0\,1\,0\,1 = x^9 + \quad x^6 + x^5 + x^2 + 1 \; mod\, G = x^4 + x^2 + x$
Fall 3 : $N'_{Fehler} = 1\,\mathbf{0}\,\mathbf{1}\,\mathbf{0}\,1\,0\,0\,1\,\mathbf{1}\,1 = x^9 + \quad x^7 + x^5 + x^2 + x + 1 \; mod\, G = 0$

Im ersten Fall handelt es sich um die richtige Nachricht, im zweiten und dritten Fall nicht.Die Nachricht im dritten Fall unterscheiden sich an 4 Stellen der Bitfolge. Das Generatorpolynom G teilt aber das Polynom $x^9 + x^7 + x^5 + x^2 + x + 1$, deshalb ist die Prüfsumme wieder gleich 0. Der Empfänger kann nicht feststellen, dass ein Fehler stattgefunden hat, weil er bei dem Prüfsumme = 0 davon ausgehen muss, dass wohl alles in Ordnung ist. Das Ziel dieses Verfahrens ist es auch, den 3. Fall zu vermeiden.

Wir zerlegen jetzt die falsche Nachricht N'_{Fehler} im dritten Fall in zwei Teilen. Einmal das richtige Polynom N' was eigentlich versendet wurde, und die Bitfolge e, an deren Stellen es Fehler gegeben hat.

$$\begin{aligned} N'_{Fehler} &= 1\,\mathbf{0}\,\mathbf{1}\,\mathbf{0}\,1\,0\,0\,1\,\mathbf{1}\,1 \\ N' &= 1\,\mathbf{1}\,0\,1\,1\,0\,0\,1\,0\,1 \\ e &= 0\,\mathbf{1}\,\mathbf{1}\,\mathbf{1}\,0\,0\,0\,0\,\mathbf{1}\,0 \end{aligned}$$

Wenn man eine bitweise XOR Operation auf N' und e anwendet, dan erhält man wieder N'_{Fehler}. Wir wissen, dass G ein Teiler von N'_{Fehler} ist, also $G \,|\, N'_{Fehler}$. Aus $N'_{Fehler} = N' + e$ folgt, dass $G \,|\, (N' + e)$. Wir wissen auch, dass $G \,|\, N'$. Daraus folgt, dass $G \,|\, e$.Das ist die Sache, die wir vermeiden möchten. Das heißt, wir können jetzt Fehlerpolynomen aussuchen, die wir erkennen möchten, und die dürfen alle nicht Vielfache von G sein. Das ist das was man finden muss, wenn man ein gutes G haben möchte.

Nun betrachten wir die erste Anforderung an das Generatorpolynom G genauer. Wenn pq das Produkt von 2 Polynomen p und q, dann kann pq nur dann ein Monom x^k sein, wenn p und q beide Monome sind. Anders ausgedrückt, wenn unser Generatorpolynom G selbst kein Monom ist, dann kann G auch das Fehlerpolynom e nicht teilen, wenn e ein Monom ist.

Die zweite Art von Fehler, die man gerne erkennen möchte, ist die sogenannten Bündelfehler. Dabei geht es darum, dass eine ganze Reihe von Fehlern in einem räumlich begrenzten Bereich auftreten. Hier nehmen wir einen Bündelfehler der Länge 4. Also $e = \underline{\mathbf{1}\,0\,\mathbf{1}\,\mathbf{1}}\,0\,0 = x^5 + x^3 + x^2 = (x^3 + x + 1) \cdot x^2$.Das muss aber nicht heißen, dass alle 4 Stellen fehler sein müssen. Die faktoren $(x^3 + x + 1)$ und x^2 sind teilerfremd,das heißt, wenn $G \,|\, e$ ist, dann muss $G \,|\, (x^3 + x + 1)$, weil wir oben schon ausgeschlossen, dass G kein Monom ist. Das geht aber nur, wenn der Grad von G höchstens 3 ist. Also die zweite Forderung die wir formulieren können ist, dass G möglichst hohen Grad haben soll, damit dann Bündelfehler erkannt werden können, die nicht breiter als der Grad von G ist. Aber je höher der Grad des Generatorpolynoms ist, desto höher ist auch natürlich

die Redundanz, da man mehr Stellen hinter der Nachricht als Prüfbits hängen muss.

Die dritte wünschenswerte Eigenschaft des Generatorpolynoms, die man gerne hätte,ist, dass G möglichst viele Polynome der Form $x^k + 1$ nicht teilen,damit Fehler, die aus 2 Bits bestehen, deren Abstand nicht zu größ ist, erkannt werden können. Zum Beispiel wenn das Fehlerpolynom $e = 1\,0\,0\,0\,0\,0\,0\,0\,1$ ist.Ein Beispiel aus der Praxis für das Generatorpolynom, das häufig Im Ethernet Protokoll (IEEE802.3) eingesetzt wird, ist das Polynom $G = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$.

3.2 Testing

Wir haben eine Methode geschrieben

```
void test(size_t len, const char msg[len], uint32_t generator, int imp)
```

,um zu testen, ob die CRC32-Methoden wie erwartet ordnungsgemäß funktionieren.Das **int imp** ermöglicht es dem Benutzer, auszuwählen, welche Implementierung des CRC32-Algorithmus er testen möchte, da wir den Algorithmus auf verschiedene Arten implementiert haben.

Zuerst suchen wir nach der Prüfsumme, die durch die gegebene Nachricht und das Generatorpolynom generiert wird. Anschließend hängen wir die Prüfsumme an die Nachricht N an, wodurch eine neue Nachricht N' entsteht. Anschließend führen wir die CRC32-Operation durch, indem wir zunächst G an N' anhängen.

Wenn das Ergebnis 0 ist, liegt beim Versenden der Nachricht möglicherweise kein Fehler oder keine Datenbeschädigung vor. Das Ergebnis dieser Methode sollte jedoch immer 0 sein, da wir die Nachricht eigentlich nirgendwohin versenden und es beim Versenden keine Fehler geben sollte.

4 Performanzanalyse

In unserem Rahmenprogramm wurden insgesamt drei Implementierungen V0, V1, und V2 implementiert. Die naive Implementierung (V2) verwendet die standardmäßige Bitweise Operation ohne Optimierung. Die zweite Vergleichsimplementierung (V1) verwendet eine Lookup-Tabelle, mit vorberechneten Byte Werten für eine schnellere Laufzeit. Und die Hauptimplementierung (V0) verwendet Lookup-Table und SIMD als Optimierung.

Alle Tests werden auf einem System mit Intel I5-10300H, 2.50GHz, und mit 1 x 16GB DDR4 RAM unter 64 Bit Ubuntu 22.04.2 ausgeführt. Der Code ist mit GCC 11.3.0 mit der Option -O2 kompiliert.

Die Effizienz einer Implementierung kann anhand der Leistung im Vergleich zur Rechengeschwindigkeit bewertet werden. Durch den Vergleich der Laufzeit verschiedener Implementierungen mit ähnlichen Eingaben können wir deren Effizienz vergleichen.

Um die Performanz zu analysieren, haben wir zwei verschiedene Szenarien getestet: größere und kleinere Eingaben. Bei den kleineren Eingaben haben wir Intervalle von 50 Bytes versucht, während wir bei den größeren Eingaben Intervalle von 1000 Bytes verwendet haben. Für die erste Grafik wird der Code 2,000,000 mal ausgeführt, und für die zweite 200,000 mal.

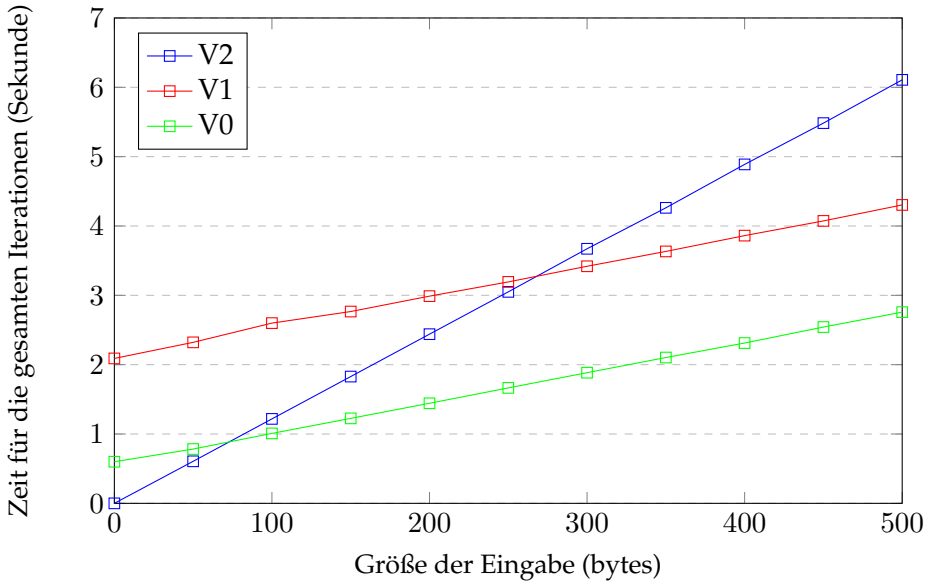

Abbildung 6: Vergleich zwischen 3 Implementierungen von CRC32 mit 2.000.000 Iterationen

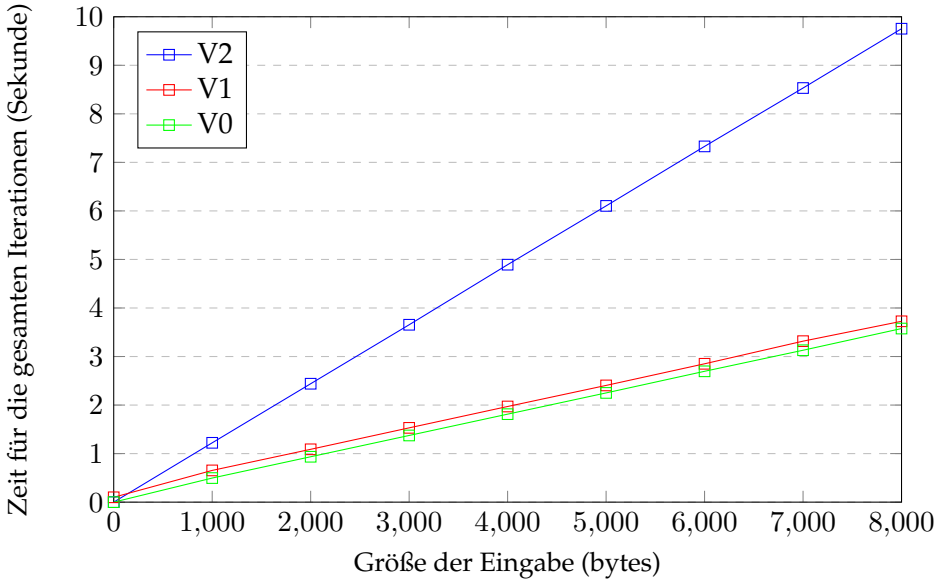

Abbildung 7: Vergleich zwischen 3 Implementierungen von CRC32 mit 200.000 Iterationen

Auf der einen Seite kann man laut der ersten Grafik deutlich sehen, dass alle drei Implementierungen stetig gestiegen sind. Bei 500 bytes Eingabe braucht V0 die geringste Zeit, nämlich fast 3 Sekunden. Das ist 2 mal schneller im Vergleich zu V2, die für gleiche Eingabe knapp 6 Sekunden braucht. Auffällig ist es, dass V0 am Anfang des Diagramms schneller ist als die Anderen. Dies könnte passieren, weil das Erstellen einer Tabelle ein Overhead ist. Das Füllen der Tabelle erzwingt zusätzlichen Speicherplatz und Zeit. Die Tabelle wird dadurch 256 Zahlen haben, von denen die meisten unnötig für kleinere Eingaben sind. V0 erfordert auch viele Speicherzugriffe, welche bei kleinen Eingaben erheblich sein können und sich auf die Leistung schlecht auswirken.

Auf der anderen Seite ist V0, das eine Lookup-Tabelle und SIMD verwendet, etwa 0,2 Sekunden schneller als V1. Allerdings ist der Unterschied nicht so signifikant, und das aus mehreren Gründen. Einer davon ist, dass die Berechnung jedes Bytes voneinander abhängig ist und daher nicht parallel ausgeführt werden kann. Aus diesem Grund können wir SIMD nur in einem kleinen Teil des Codes verwenden und nämlich zur Berechnung der Lookup-Tabelle.

Anhand des vorliegenden Graphen lässt sich deutlich erkennen, dass V2 oder die Naive-Implementierung wesentlich langsamer ist als V0 und V1. Dieser Geschwindigkeitsunterschied resultiert daraus, dass V2 wiederholt viele Bytes mit denselben Werten berechnet, während V0 und V1 alle Werte (0x00 - 0xFF) lediglich einmal in der Lookup-Table berechnen. Demzufolge erfordert V2 eine deutlich längere Laufzeit im Vergleich zu den anderen Implementierungen. Nach einer detaillierten Untersuchung des Graphen sind wir zu dem Schluss gekommen, dass unsere Hauptimplementierung mit Lookup-Table und SIMD die optimale Wahl für die Hauptimplementierung darstellt.

5 Zusammenfassung und Ausblick

Die Ausarbeitung präsentiert drei verschiedene Implementierungen des CRC32 Algorithmus: den algebraischen Ansatz, den bitorientierten Ansatz und den tabellengesteuerten Ansatz. Der algebraische Ansatz basiert auf der mathematischen Definition des CRC und führt eine Polynomdivision durch. Der bitorientierte Ansatz führt die Polynomdivision direkt mit den Bits der Eingangsdaten durch, während der tabellengesteuerte Ansatz eine Lookup-Tabelle verwendet, um die teuren XOR-Operationen zu vermeiden.

Es wurden auch wichtige Aspekte bei der Wahl eines geeigneten Generatorpolynoms für den CRC32-Algorithmus diskutiert. Ein gutes Generatorpolynom sollte eine hohe Wahrscheinlichkeit haben, Fehler zu erkennen, und keine Polynome der Form $x^k + 1$ teilen können, um Bündelfehler zu erkennen.Die Korrektheit des CRC32-Algorithmus wird ebenfalls betrachtet, wobei die deterministische Erzeugung des Prüfwerts und die zuverlässige Erkennung von Fehlern im Fokus stehen.

Abschließend werden Performanzanalysen der Implementierungen durchgeführt. Dabei werden verschiedene Szenarien mit unterschiedlichen Eingabegrößen getestet. Die Implementierung mit der Lookup-Tabelle und SIMD-Optimierung zeigt die beste Performanz und ist effizienter als die anderen Implementierungen.

Zusammenfassend lässt sich sagen, dass der CRC32-Algorithmus recht schnell und zuverlässig ist, um zu überprüfen, ob beim Senden einer Nachricht ein Fehler aufgetreten ist. Aus diesem Grund wird dieser Algorithmus bis heute beispielsweise im Ethernet-Protokoll verwendet.

Andere Ansätze zur Optimierung des CRC32 Algorithmus können untersucht werden,um die Geschwindigkeit und die Fähigkeit zur Fehlererkennung des Programms zu verbessern.

Literatur

[1] Martin Stigge, Henryk Plötz, Wolf Müller, and Jens-Peter Redlich. Reversing crc–theory and practice. *Berlin: Humboldt University Berlin*, 17, 2006. `https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-05/SAR-PR-2006-05_.pdf` , visited 2023-06-16.

[2] Ross Williams et al. A painless guide to crc error detection algorithms. *Internet publication, August*, 75, 1993. `http://www.ross.net/crc/download/crc_v3.txt`, visited 2023-06-16.