



UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Studienarbeit (DPO4)

RAM-Modell und 64Bit Prozessoren

Alexander Spot

April 2008

Betreuer: PD Dr. habil. Martin Ziegler

Vorbemerkung

Der Artikel "ON FASTER INTEGER CALCULATIONS USING NON-ARITHMETIC PRIMITIVES" von Katharina Lürwer-Brüggemeier und Martin Ziegler bildet die Hauptgrundlage für diese Studienarbeit. Insbesondere die in Abschnitt 4 des Artikels aufgeworfenen Fragen sind Gegenstand dieser Arbeit.

Inhaltsverzeichnis

1	Einleitung	5
2	Aufgabenstellung	6
3	Algorithmen zur Polynomauswertung	6
3.1	Bshoutys Algorithmus	7
3.1.1	Funktionsweise	8
3.2	Horner-Schema	10
3.3	Look-up Tabellen	10
4	Implementierung des Benchmarks	10
5	Implementierung der Algorithmen	11
5.1	Datenstruktur	11
5.2	Bshouty	12
5.3	Horner-Schema	15
5.4	Look-up Tabelle	15
5.5	Theoretische Laufzeit der Assembler Codes	16
6	Messverfahren	18
6.1	systematische Fehler	19
7	Anwendung Benchmark	19
8	Testumgebung	22
9	Testdurchführung	22
10	Auswertungen	23
10.1	Laufzeit bei einem fest vorgegebenen Polynom	23
10.2	Laufzeit der Einzeloperationen bei BSHOUTY	24
10.3	Laufzeit bei steigendem Grad eines Polynoms	26
10.3.1	Laufzeit bei steigendem Definitionsbereichs X bei dem Polynom $P = x^2 + x + 1$	29
10.4	Einfluss der Auswertestelle	34
10.5	Einfluss des Prozessor Cache	35
10.6	Einfluss des Prozessor Architektur	38

11 Fehlerbetrachtung	38
11.1 Standardabweichung	39
11.2 Lineare Regression	39
11.3 Fehlerabschätzung der Zeitmessung	40
12 Fazit und Ausblick	40

1 Einleitung

In der modernen Welt existieren viele Problemstellungen, die sich durch eine Auswertung von einem oder mehreren Polynomen lösen lassen. Als Beispiele seien hier die Reed-Solomon Codes und die Spline-Interpolation genannt.

Um Polynome auszuwerten, sind etliche Methoden erdacht worden. Eine triviale Möglichkeit wäre es, jeden einzelnen Koeffizient mit seiner dazugehörigen Potenz von x zu multiplizieren und anschließend die Zwischenergebnisse zu addieren. Das erfordert aber die Speicherung der Zwischenergebnisse. Zudem hängt die Anzahl der Zwischenergebnisse vom Grad d des Polynoms ab: je höher der Grad ist, desto mehr Ergebnisse müssen gespeichert werden. Auch werden die Potenzen von x wiederholt berechnet. Diese Nachteile lassen sich mit dem Horner-Schema umgehen. Dort findet eine geschickte Reihenfolge von Multiplikationen und Additionen statt, siehe dazu Abschnitt 3.2.

Eine weitere Möglichkeit ist der Algorithmus von Bshouty. Mit diesem lassen sich ganzzahlige Polynome beliebigen Grades mit konstant vielen Operationen auswerten. Im Voraus muss allerdings das Intervall festgelegt werden, in dem die Auswertungen stattfinden sollen. Der Definitionsbereich muss also vorher festgelegt worden sein.

Da bei Bshoutys Algorithmus die Polynomauswertung mit konstant viele Operationen möglich ist, besitzt das Verfahren auch eine konstante Laufzeit, sofern die Größe der in den einzelnen Zwischenschritten erzeugten Ergebnisse keine Rolle spielt. Das ist aber mit den heutigen Computern nur beschränkt umsetzbar. Denn dort steht nur eine endliche Rechengenauigkeit zur Verfügung. Mit den neueren Prozessorgenerationen, die den *x86-64* Befehlssatz unterstützen, ist die Rechengenauigkeit von 32Bit auf 64Bit zwar vergrößert worden.

Doch reicht das aus, um Bshouty auf diesen Prozessoren praxisnah einzusetzen? Sind andere Verfahren wie Horner-Schema oder Look-up Tabellen Bshoutys Algorithmus unter- oder überlegen oder gar gleichwertig? Das führt zur der Fragestellung, die in dieser Studienarbeit beantwortet wird:

Ist Bshoutys Algorithmus mittlerweile praktikabel einsetzbar?

2 Aufgabenstellung

Um die Praxistauglichkeit von Bshoutys Algorithmus zu ermitteln, werden die folgenden Methoden zur Auswertung ganzzahliger Polynome als Referenz herangezogen:

- Bshoutys Algorithmus
- Horner-Schema
- Look-up Tabellen

Die Leistungsfähigkeit der verschiedenen Verfahren werden an Hand von Tests bestimmt. Im einzelnen sind das:

1. Messung der Laufzeit bei einem fest vorgegebenen Polynom P
2. Messung der Laufzeit der Einzeloperationen bei Bshoutys Algorithmus
3. Messung der Laufzeit bei steigendem Grad d eines Polynoms P
4. Bestimmung des Einflusses der Auswertung an konstanten bzw. variablen Stellen
5. Bestimmung des Einflusses des Prozessor Caches
6. Bestimmung des Einflusses der Prozessor Architektur

Bei Punkt 1 wird die Laufzeit gemessen, die die jeweiligen Verfahren bei der Auswertung bei einem festen Polynom P benötigen. Bei Punkt 2 wird untersucht, ob es einen Befehl gibt, der Bshoutys Algorithmus in seiner Ausführung verzögert. Punkt 3 beschäftigt sich mit der Frage, ob Bshoutys Algorithmus tatsächlich konstant ist und wie die anderen Verfahren auf den steigenden Grad des Polynoms P reagieren.

Ist es relevant, ob wiederholt an derselben Stelle das Polynom ausgewertet wird? Oder ist es egal, wenn die Auswertestelle variiert? Diesen Fragen geht Punkt 4 nach. Welche Bedeutung hat der Prozessor Cache auf die Algorithmen? Diese Frage soll Punkt 5 klären. Zum Schluss wird getestet, inwieweit die PC Architektur die Laufzeit der Algorithmen beeinflusst.

Um die Tests durchzuführen, ist ein geeignetes Programm erforderlich. Dieses zu erstellen, war ebenfalls Bestandteil der Aufgabe. Der Abschnitt 4 auf Seite 10 geht näher auf die Aspekte der Realisierung ein.

3 Algorithmen zur Polynomauswertung

In diesem Abschnitt wird die Funktionsweise der Algorithmen vorgestellt. Gegeben sei:

$$P := \sum_{i=0}^d p_i x^i \quad (1)$$

mit:

$$p_i \in \mathbb{N}^{\geq 0} \quad (2)$$

Wie in der Einleitung auf Seite 5 geschildert, gilt es noch den Definitionsbereich X vorher anzugeben:

$$X \in \mathbb{N} \quad (3)$$

Weiter muss gelten:

$$x \in \{0, 1, \dots, X\} \quad (4)$$

3.1 Bshoutys Algorithmus

Bei Bshoutys Algorithmus muss der Wert Z vorberechnet werden:

$$Z > \max \{X^d * \rho, (X^d + 1) * X\} \quad (5)$$

mit

$$\rho := \sum_{i=0}^d p_i \quad (6)$$

Zusätzlich muss man noch $P(Z)$ berechnen. Dann lässt sich das Polynom P an der Stelle x auswerten:

1. $a = \left\lfloor \frac{Z^{d+1}}{Z-x} \right\rfloor$
2. $b = a * P(Z)$
3. $c = \left\lfloor \frac{b}{Z^a} \right\rfloor$
4. $result \equiv c \text{ mod } Z$

Durch Vorbereitung von Z , Z^{d+1} , Z^d und $P(Z)$ ist die Laufzeit von Bshoutys Algorithmus konstant.

3.1.1 Funktionsweise

Die Korrektheit ist im eingangs erwähntem Artikel enthalten [LBZ07] und wird hier skizziert. Schritt 1 berechnet:

$$\left\lfloor \frac{Z^{d+1}}{Z-x} \right\rfloor \quad (7)$$

$$= \left\lfloor \frac{Z * Z^d}{Z * (1 - (x/Z))} \right\rfloor \quad (8)$$

$$= \left\lfloor Z^d * \frac{1}{1 - (x/Z)} \right\rfloor \quad (9)$$

Da $Z \gg x$ ist, kann man mittels der Geometrische Reihe den Ausdruck in der Klammern ersetzen durch:

$$= \left\lfloor Z^d * \sum_{i=0}^{\infty} \left(\frac{x}{Z}\right)^i \right\rfloor \quad (10)$$

$$= \left\lfloor Z^d * \sum_{i=0}^d \left(\frac{x}{Z}\right)^i + Z^d * \sum_{i=d+1}^{\infty} \left(\frac{x}{Z}\right)^i \right\rfloor \quad (11)$$

Nach Ausklammern von x^{d+1} und $(Z^{d+1})^{-1}$ in der zweiten Summe erhält man:

$$\left\lfloor Z^d * \sum_{i=0}^d \left(\frac{x}{Z}\right)^i + \left(\frac{x^{d+1}}{Z}\right) * \sum_{i=0}^{\infty} \left(\frac{x}{Z}\right)^i \right\rfloor \quad (12)$$

Da $Z \gg x$ ist, kann man mittels der Geometrische Reihe die **zweite** Summe ersetzen durch:

$$\left\lfloor Z^d * \sum_{i=0}^d \left(\frac{x}{Z}\right)^i + \left(\frac{x^{d+1}}{Z}\right) * \left(\frac{1}{1 - (x/Z)}\right) \right\rfloor \quad (13)$$

$$= \left\lfloor Z^d * \sum_{i=0}^d \left(\frac{x}{Z}\right)^i + \left(\frac{x^{d+1}}{Z-x}\right) \right\rfloor \quad (14)$$

$$= \left\lfloor \underbrace{Z^d + Z^{d-1}x + Z^{d-2}x^2 + \dots + Zx^{d-1} + x^d}_{\in \mathbb{N}} + \underbrace{\left(\frac{x^{d+1}}{Z-x}\right)}_{< 1} \right\rfloor \quad (15)$$

Da $Z > (x^d + 1) * x \Leftrightarrow Z - x > x^{d+1}$ ist, ist der Ausdruck in der Klammer kleiner 1. Dadurch dass die Summe abgerundet wird, trägt der Ausdruck in der Klammer nicht mehr zu der Summe mit bei. Es gilt somit:

$$\left\lfloor \frac{Z^{d+1}}{Z-x} \right\rfloor = Z^d + Z^{d-1}x + Z^{d-2}x^2 + \dots + Zx^{d-1} + x^d \quad (16)$$

Die Koeffizienten x^i der Summe in Gleichung 16 kann man als Stellen einer Z -adischen Zahl auffassen. Die Koeffizienten des Polynoms P kann man ebenso als Stellen einer Z -adischen Zahl interpretieren:

$$Z^d p_d + Z^{d-1} p_{d-1} + Z^{d-2} p_{d-2} + \dots + Z p_1 + p_0 \quad (17)$$

Auf diese Weise erhält man:

$$P(Z) = Z^d p_d + Z^{d-1} p_{d-1} + Z^{d-2} p_{d-2} + \dots + Z p_1 + p_0 \quad (18)$$

Multipliziert man diese beiden Summen - wie in Schritt 2. durchgeführt wird -, so erhält man eine Z -adische Zahl mit Z^{2d} Stellen. Die Abbildung 1 veranschaulicht Schritt 2. Dabei steht an Stelle Z^d das Ergebnis $P(x)$. Um an dieses zu gelangen, genügt es, das Produkt durch Z^d zu dividieren (Schritt 3) und die vorderen Stellen mit der Modulo-Operation abzuschneiden (Schritt 4).

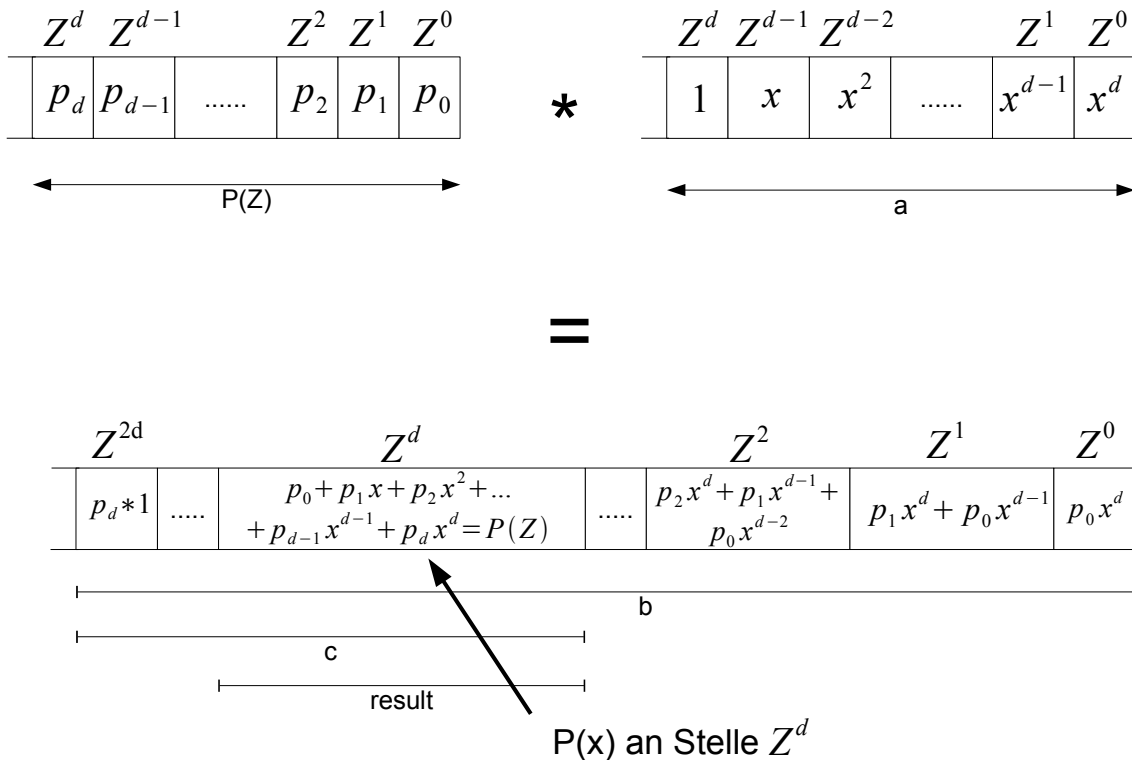


Abbildung 1: Bshoutys Algorithmus: Schritte 2,3 und 4 veranschaulicht

3.2 Horner-Schema

Mit dem Horner-Schema wertet man das Polynom P wie folgt aus:

$$\sum_{i=0}^d p_i x^i = (((p_d * x + p_{d-1}) * x + p_{d-2}) * x + \dots) * x + p_0 \quad (19)$$

Zuerst multipliziert man p_d mit x , addiert dazu dann p_{d-1} . Das Ergebnis multipliziert man wieder mit x und addiert diesmal p_{d-2} dazu. Sukzessiv fährt man auf diese Art fort bis zu dem letzten Koeffizienten p_0 .

Zu beachten ist, dass die Anzahl der Multiplikationen und Additionen von der Größe des Grads d abhängt. Hat das Polynom Grad d , müssen auch genau d Multiplikationen und Additionen durchgeführt werden. Die Laufzeit hängt also linear vom Grad des Polynoms ab.

3.3 Look-up Tabellen

Bei Look-up Tabellen werden sämtliche Ergebnisse im vorher festgelegten Definitionsbereich X vorberechnet. Bei der Auswertung des Polynoms P an der Stelle x wird dann einfach bei dem entsprechenden Eintrag nachgesehen. Der dort gefundene Wert wird zurückgegeben. Die Laufzeit ist somit konstant. Die Vorberechnung geht dabei nicht in die Betrachtung der Leistungsfähigkeit der Look-up Tabellen mit ein.

4 Implementierung des Benchmarks

Der zweite Schritt von Bshoutys Algorithmus führt eine Multiplikation mit beliebig großer Genauigkeit durch, d.h. die Größe des Produktes kann sich mitunter verdoppeln: Bei einer Multiplikation mit zwei 64Bit großen Faktoren kann das Produkt bis zu 128Bit groß werden. Umgekehrt verhält es sich bei der Division - wie in Schritt 1: Ist hier der Dividend 128Bit groß und wird durch einen 64Bit großen Divisor geteilt, so wird der Quotient maximal 64Bit groß.

Genau diese Art der arithmetischen Operationen werden von den Prozessoren mit dem Befehlssatz *x86_64* angeboten. Diese bieten die Möglichkeit, mit nur einem Befehl Multiplikation und Division mit bis zu 128Bit Genauigkeit durchzuführen [AMD07]. Diese Befehle sind aber nicht aus Hochsprachen wie C/C++ direkt aufrufbar.

Aus diesem Grund erfolgt die Implementierung in Assembler. Wäre aber das Benchmark komplett in Assembler geschrieben, hätte sich nur der Entwicklung des Benchmarks verzögert. Sowohl die Fehleranfälligkeit als auch die Wartbarkeit hätten sich drastisch erhöht. Daher sind unwesentliche Teile, wie die Benutzereingabe, in C geschrieben. Der relevante Teil erfolgt aber in Assembler. Somit besteht der Benchmark im wesentlichen aus C mit Inline-Assembler.

Mit der *GNU-Compiler-Collection (GCC)* lassen sich Assembler Sequenzen direkt im C/C++-Quellcode einbetten. Dafür existiert ein eigenes Sprachkonstrukt `__asm__` mit folgender Struktur:

```

__asm__ ( "Assembler Befehle"
: Ausgabe (optional)
: Eingabe (optional)
: Liste von "clobbered" Registern (optional)
);

```

Für weitere Informationen über Inline-Assembler mit dem gcc, sei hier auf das "*GCC-Inline-Assembly-HOWTO*" verwiesen [GCC]. Konkrete Anwendungen sind aber auch hier zu sehen. Sie befinden sich im folgenden Abschnitt 5.

5 Implementierung der Algorithmen

In diesem Abschnitt werden die verschiedenen Implementierungen der Algorithmen vorgestellt. Hier ist zu beachten, dass die gezeigten Assembler Befehle ausschließlich für eine 64Bit Umgebung geeignet sind. Es ist allerdings ohne viel Aufwand möglich, diese 64Bit Befehle auf eine 32Bit Umgebung zu portieren. Dazu muss das Suffix '*q*' am Ende der 64Bit Befehle durch das Suffix '*l*' ersetzt werden. Das Präfix '*r*' in den Register-Namen muss durch das Präfix '*e*' ersetzt werden. So wird zum Beispiel aus dem 64Bit Befehl "*divq %rsi*" der 32Bit Befehl "*divl %esi*".

5.1 Datenstruktur

Um das Polynom und die verschiedenen vorberechneten Werte effizient abzuspeichern, habe ich folgende Datenstruktur eingesetzt:

```

1 struct polynom
2 {
3     unsigned long x;           // Auswertstelle x
4     unsigned long X;         // Definitionsbereich
5     unsigned long d;         // Grad des Polynoms
6     unsigned long Z;         // <= Z
7     unsigned long pZ;        // <= P(Z)
8     unsigned long Zm0;       // <= Z - 1
9     unsigned long Zdp_low;   // untere 64 Bits von Z^(d+1)
10    unsigned long Zdp_high;  // obere 64 Bits von Z^(d+1)
11    unsigned int zd;         // <= d*log2(Z)
12    unsigned long Zmx;       // <= Z - x
13    int overflow;           // Flag
14    unsigned long *A;        // Koeffizienten p_i
15    unsigned long *Table;    // Werte der Look-up Tabelle
16 };

```

Abbildung 2: Datenstruktur zur Speicherung des Polynoms

Die meisten Einträge sind durch die Kommentare bereits gekennzeichnet, nur die Werte zd , ZmO und $overflow$ sind noch unerklärt. zd und ZmO werden im Abschnitt 5.2 erklärt. $overflow$ kennzeichnet den Überlauf bei $P(Z)$. Falls der Wert $P(Z)$ die 64Bit Grenze überschreitet, wird $overflow$ gesetzt. Auch falls $d * \log_2(Z) > 64$, wird $overflow$ gesetzt. Ist $overflow$ gesetzt, so wird Bshoutys Algorithmus nicht durchgeführt, da das Ergebnis mit Sicherheit falsch ist. Ansonsten hat $overflow$ keinen Einfluss auf Bshoutys Algorithmus.

5.2 Bshouty

Die Implementierung von Bshoutys Algorithmus beinhaltet noch die Modifikation, dass Z zur nächst größeren 2er-Potenz aufgerundet wird. Es gilt somit:

$$Z > \max \{X^d * \rho, (X^d + 1) * X\} \wedge Z = 2^z \quad (20)$$

mit

$$z \in \mathbb{N}^{>0}$$

Dadurch entstehen einige Vorteile und Nachteile. Nachteilig wirkt sich die künstliche Vergrößerung von Z aus, dass $P(Z)$ schon die 64Bit Rechengenauigkeit überschreitet, obwohl auch ein kleineres Z geeignet wäre, wie in diesem Beispiel zu sehen ist:

$$P = x^5$$

mit

$$X = 4, x = 4$$

Es folgt:

$$\begin{aligned} Z &> \max \{4^5 * 1, (4^5 + 1) * 4\} = \max \{1024, 4100\} \\ &\Rightarrow Z > 4100 > 4096 = 2^{12} \end{aligned}$$

Da Z eine 2er-Potenz sein muss, wird Z aufgerundet auf:

$$Z = 8192 = 2^{13}$$

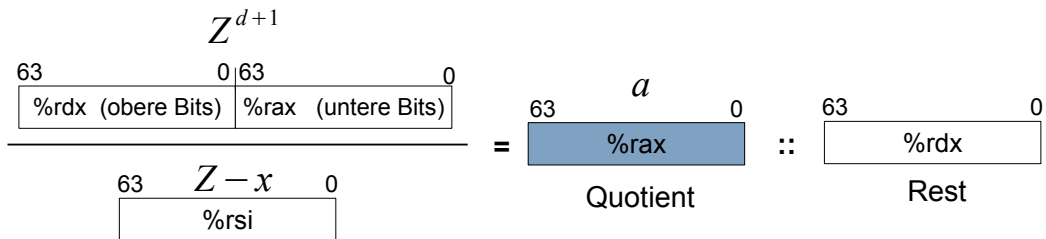
Daraus folgt aber, dass:

$$P(Z) = (2^{13})^5 = 2^{13*5} = 2^{65} > 2^{64}$$

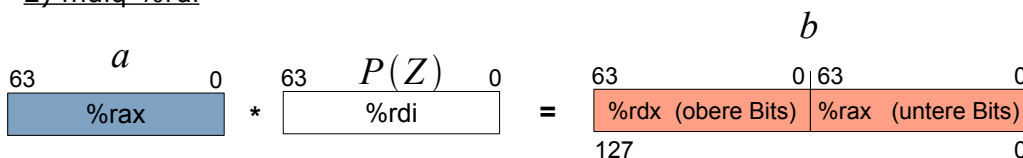
Hier wird also mit $P(Z)$ die 64Bit Rechengenauigkeit überschritten.

Aber vom großen Vorteil ist, dass die zweite Division $c = \lfloor \frac{b}{Z^d} \rfloor$ durch einen *SHIFT*-Befehl ersetzt werden kann. Die letzte Operation $result \equiv c \bmod Z$ kann dadurch auch durch einen *AND*-Befehl ersetzt werden. Die folgende Abbildung 3 veranschaulichen die Vorgehensweise:

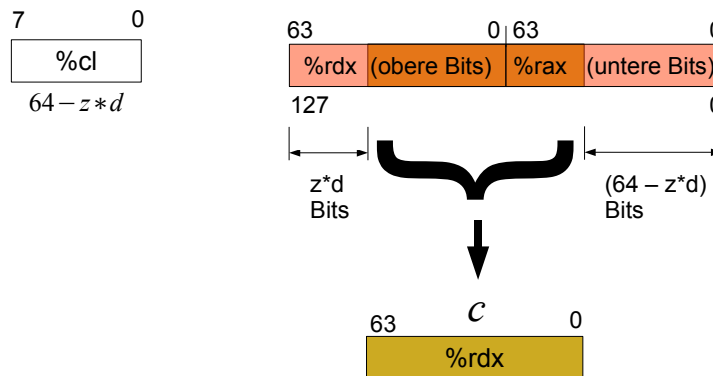
1) `divq %rsi`



2) `mulq %rdi`



3) `shld %cl, %rax, %rdx`



4) `andq %rbx, %rdx`

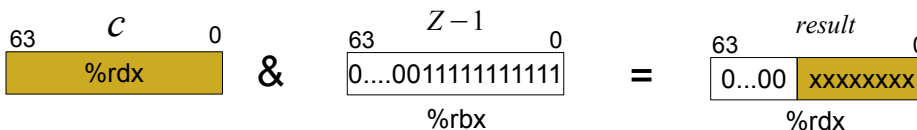


Abbildung 3: Bshoutys Algorithmus in Assembler - veranschaulicht

Die erste Operation ist die Division. Hier werden die beiden Register $\%rdx$ und $\%rax$ zusammengefasst und durch den Wert im Register $\%rsi$ geteilt. Der Quotient a ist daraufhin im Register $\%rax$, während der Rest im Register $\%rdx$ gespeichert wird. Da der Rest für die weiteren Schritte nicht mehr benötigt wird, kann das Register $\%rdx$ durch andere Operationen überschrieben werden. Genau das geschieht bei der darauf folgenden Multiplikation.

Das Register $\%rdx$ enthält nun die oberen 64Bit der Multiplikation und Register $\%rax$ die unteren 64Bit. Im Original Algorithmus erfolgt jetzt die zweite Division. Da aber Z eine 2^z -Potenz ist, können die Bits in den beiden Registern um zd Bits nach rechts verschoben werden. Auf diese Weise wird die Division elegant umgesetzt. Z kann sehr schnell sehr groß werden. Daher müssten auch zd viele Bits nach rechts verschoben werden, um das Ergebnis im Register $\%rax$ zu erhalten. Hier bedient man eines Tricks und verschiebt nur die oberen 64Bit, also Register $\%rdx$, um $(64 - zd)$ Bits nach links und füllt die verbleibenden Bits mit den oberen Bits des Registers $\%rax$ auf. Um zd nicht jedes Mal neu zuberechnen, ist der Wert vorberechnet und in der Datenstruktur abgespeichert.

Der letzte verbleibende Schritt setzt die Modulo-Operation um. Wiederum, da Z eine 2^z -Potenz ist, kann man die Modulo-Operation auf eine $\&$ -Verknüpfung reduzieren. Dazu subtrahiert man 1 von Z und erhält somit, dass alle Bits rechts neben der 1 gesetzt werden. Die ursprüngliche 1 wird dabei gelöscht. Nur in dem Bereich mit den gesetzten Bits werden die Bits von $\%rdx$ bei der AND-Operation dann übernommen. Die Modulo-Operation ist umgesetzt und Bshoutys Algorithmus ist beendet. Das Resultat steht in Register $\%rdx$ und ist maximal $Z - 1$ groß.

Die genaue Implementation ist in Abbildung 4 zu sehen:

```

1 static inline long bshouty(struct polynom *poly)
2 {
3     long result;
4
5     if (poly->overflow == FALSE)
6     {
7         __asm__ __volatile__ ("## BSHOUTY ##\n\t"
8             "divq %%rsi\n\t"
9             "mulq %%rdi\n\t"
10            "shld %%cl, %%rax, %%rdx\n\t"
11            "andq %%rbx, %%rdx"
12
13            : "=d" (result)
14            : "S" (poly->Z - poly->x), "D" (poly->pZ), "d" (poly->Zdp_high), "a"
15            (poly->Zdp_low), "b" (poly->Zm0), "c" (MaxBits - poly->zd)
16            //      : no clobbered register ["%eax", "%ebx", "%ecx", "%edx", "%esi", "%edi"]
17            );
18
19        return result;
20    }
21
22    return -1;
23 }

```

Abbildung 4: Implementierung Bshoutys Algorithmus mit Inline-Assembler

5.3 Horner-Schema

Um das Horner-Schema effizient zu implementieren, kann man sich des Compilers bedienen. Dazu wurde das Horner-Schema in C geschrieben. Der Compiler generiert anschließend daraus den Assembler Code. Der Assembler Code ist in Abbildung 5 zu sehen.

```
1 static inline long asm_horner(struct polynom *poly)
2 {
3     long result;
4
5     __asm__ __volatile__ ("## ASM_HORNER ##\n\t"
6         "movq (%rsi), %rdi\n\t"
7         "testq %rbx, %rbx\n\t"
8         "je 0f\n\t"
9         "movl $1, %%eax\n"
10        "1: \n\t"
11        "imulq %%rcx, %rdi\n\t"
12        "addq (%rsi, %rax, 8), %rdi\n\t"
13        "inc %%rax\n\t"
14        "cmpq %rax, %rbx\n\t"
15        "jae 1b\n"
16        "0:\n\t"
17
18        : "=D" (result)
19        : "S" (poly->A), "c" (poly->x), "b" (poly->d)
20        : "%eax", "%edx" // clobbered register
21        );
22
23     return result;
24 }
```

Abbildung 5: Implementierung Horner-Schema mit Inline-Assembler

Zuerst wird der Koeffizient p_d nach Register $\%rdi$ kopiert. Anschließend erfolgt der Test, ob der Grad des Polynoms gleich oder größer als 1 ist. Falls nicht, dann erfolgt sofort der Sprung zum Ende des Code-Abschnitts nach Zeile 16. Das Ergebnis steht im Register $\%rdi$.

Andernfalls wird die Schleifenvariable $count$ im Register $\%eax$ auf den Wert 1 gesetzt. Im Register $\%rcx$ ist der Wert x enthalten. Dieser wird bei der gesamten Berechnung nicht verändert und bei der Multiplikation in Zeile 11 mit dem Wert in Register $\%rdi$ multipliziert. Es folgt die Addition mit dem jeweilig passendem Koeffizienten p_i .

Damit ist der Schleifenrumpf beendet. Zeilen 13 bis 15 führen den Test durch, ob der Schleifenrumpf nochmal durchlaufen werden soll. Zum Schluss steht das Ergebnis im Register $\%rdi$.

5.4 Look-up Tabelle

Die Implementierung von Look-up Tabellen ist erheblich kürzer; bestehen sie doch nur aus einer einzigen Zeile Assembler Code. Hier muss nur noch der passende Eintrag aus der Tabelle ausgelesen werden, wie in Abbildung 6 zu sehen ist:

```

1 static inline long look_up_table(struct polynom *poly)
2 {
3     long result;
4
5     __asm__ __volatile__ ("## LOOK-UP TABLE ##\n\t"
6         "movq (%rbx, %%rcx, 8), %%rdx"
7
8         : "=d" (result)
9         : "b" (poly->Table), "c" (poly->x)
10        : "%eax", "%esi", "%edi" // clobbered register
11        );
12
13    return result;
14 }

```

Abbildung 6: Implementierung Look-up Tabelle mit Inline-Assembler

Nach Auslesen des entsprechenden Eintrags, wird das Ergebnis nach Register *%rdx* kopiert.

5.5 Theoretische Laufzeit der Assembler Codes

Aus der Tabelle 13 (Integer Instructions) in [AMD05] lassen sich für die AMD-Architektur die theoretischen Laufzeitverzögerungen der jeweiligen Assembler Implementierungen bestimmen.

In den Tabellen sind jeweils die Assembler Codes für die 64Bit Umgebung angegeben. Die Vorgehensweise, wie man die 64Bit Befehle auf eine 32Bit Umgebung portieren kann, ist im Abschnitt 5 beschrieben.

Code	Latency	
	32Bit	64Bit
-	32Bit	64Bit
divq %rsi	39	71
mulq %rdi	3	5
shld %cl,%rax,%rdx	4	4
andq %rbx, %rdx	1	1
Σ	47	81

Tabelle 1: Laufzeitverzögerungen Bshoutys Algorithmus

Code	Latency	
	32Bit	64Bit
-	32Bit	64Bit
movq (%rsi), %rdi	3	3
testq %%rbx, %%rbx	1	1
je 0f	1	1
movl \$1, %eax	1	1
1:	-	-
imulq %rcx, %rdi	3	4
addq (%rsi, %rax, 8), %rdi	4	4
inc %rax	1	1
cmpq %rax, %rbx	1	1
jae 1b	1	1
0:	0	-
Σ	$6 + d * 10$	$6 + d * 11$

Tabelle 2: Laufzeitverzögerungen Horner-Schema

Code	Latency	
	32Bit	64Bit
-	32Bit	64Bit
movq (%rbx, %rcx, 8), %rdx	3	3
Σ	3	3

Tabelle 3: Laufzeitverzögerungen Look-up Tabelle

Latency kennzeichnet die Taktzyklen, die benötigt werden, um den jeweiligen Befehl durchzuführen. Die Taktzyklen sind Schätzwerte und können je nach Zustand des Prozessor Cache variieren.

Aus den Tabellen ist ersichtlich, dass die Look-up Tabelle und Bshoutys Algorithmus konstante Laufzeit besitzen. Allerdings verzögert der Divisionsbefehl die Laufzeit von Bshoutys Algorithmus erheblich. Jetzt wird ersichtlich, warum Z zur nächsten $2er$ -Potenz aufgerundet wird. Denn damit lässt sich die zweite Division durch eine kürzere *SHIFT*-Operation ersetzen. Beim Horner-Schema erkennt man sehr schön, dass die Laufzeit wie erwartet mit dem Grad d des Polynoms linear ansteigt. Allerdings unterscheidet sich die Steigung in der jeweiligen Umgebungen: Für die 32Bit Umgebung beträgt die Steigung 10, für die 64Bit Umgebung ist sie geringfügig größer und beträgt 11.

Sehr deutlich erkennt man, dass die Algorithmen in der 32Bit Umgebung teils erheblich schneller sind als in der 64Bit Umgebung. Das ist auch erklärbar, da ja auch mehr Bits verarbeitet werden müssen, vor allem für die Division. Nur die Operation der Look-up Tabelle ist in beiden Umgebungen gleich schnell.

Vergleicht man die Laufzeitverzögerung von Bshoutys Algorithmus und die von Horner-Schema, so stellte man fest, dass erst bei Grad $d \geq 5$ für die 32Bit Umgebung bzw. $d \geq 7$ für die 64Bit Umgebung der Algorithmus von Bshouty schneller ist.

6 Messverfahren

```
1 /// CORE
2 seed = poly->X;
3 count = 0L;
4 start = clock();
5 while(loc)
6 {
7     for (i = 0L; i < THREAD_LOOPS; i++)
8     {
9         if (x_is_const == FALSE)
10        {
11            seed = (19L * seed + 1L) & ((1L << RAND_RANGE) - 1L);
12            poly->x = ((seed >> (RAND_RANGE / 2L)) * (X + 1L))
13                >> (RAND_RANGE / 2L);
14        }
15
16        result = bshouty(poly);
17    }
18
19    count++;
20
21    state = pthread_mutex_lock(&thread_mutex);
22    loc = thread_run;
23    state = pthread_mutex_unlock(&thread_mutex);
24 }
25 differenz = clock() - start;
26 /// CORE
```

Abbildung 7: C-Code der Zeitmessung bei Bshouty Algorithmus

Aus Tabelle 1 kann man die theoretische Laufzeit für Bshoutys Algorithmus entnehmen. Sie beträgt 81 Takte. Bei einem Prozessor mit nur 1GHz Taktfrequenz beliefe sich somit die zu messende Zeitdauer auf $81ns$. Das ist zu gering, als dass man diese Zeit direkt messen könnte. Daher erfolgt die Zeitmessung durch wiederholte Aufrufe der zu testenden Funktion.

Dabei hat sich gezeigt, dass die iterativen Aufrufe in nur einer Schleife zu ungenauen Messungen führen. Bei der Schleifeniteration hängt die Messgenauigkeit von der Anzahl der Wiederholungen ab. Nach mehreren Versuchen wurde ersichtlich, dass bei circa 10^8 Durchläufen der Messfehler akzeptabel klein wurde. Zusätzlich dauerte die Messung länger, je mehr Schleifendurchläufe durchgeführt wurden.

Bei einer fest vorgegebenen Messzeit und dem Messprinzip, 1 Durchlauf der zu testenden Funktion und anschließendem Test, ob die Messzeit überschritten wurde, hat sich gezeigt, dass der Test auf Messzeitende die Ergebnisse zu stark beeinflusst. Der Test verfälscht die Messergebnisse. Daher erfolgt die Messung wie folgt:

Erst nach $N = 1000$ Schleifeniterationen wird getestet, ob die Messzeit überschritten wurde oder nicht. So ist auch gewährleistet, dass die Messungen in einer vorhersehbaren Zeit durchführbar sind.

Die Zeitmessung erfolgt mit der Funktion `clock()`, die im ANSI-Standard für *C* definiert ist. `clock()` gibt die vom aufrufenden Prozess verbrauchte Prozessorzeit zurück. Dabei wird auch die Zeitmessung angehalten, wenn der Prozess angehalten wird. Der Rückgabewert der Funktion sind Ticks. Teilt man diese durch die symbolische Konstante `CLOCKS_PER_SEC`, erhält man die verbrauchte Prozessorzeit in Sekunden.

Zu Beginn der Messung wird `start = clock()` aufgerufen. Es folgt die äußere Schleife. Die Iterationen der äußeren Schleife werden in der Variable `count` mitgezählt. Innerhalb der äußeren Schleife befindet sich dementsprechend die innere Schleife. Dort wird die zu testende Funktion $N = 1000$ -mal aufgerufen. Am Ende der äußeren Schleife findet der Test auf Messzeitende statt.

Ist die Messzeit überschritten, wird `clock()` ein weiteres mal aufgerufen: `differenz = clock() - start`. Diese Differenz ist die Gesamtzeit der Messung. Aufgrund derer wird die Zeit berechnet, die die zu testende Funktion verbraucht hat. Die Laufzeit der zu messenden Funktion lässt sich dann wie folgt bestimmen:

$$Laufzeit = \frac{differenz}{CLOCKS_PER_SEC * 1000 * count} [s] \quad (21)$$

bzw.:

$$Laufzeit = \frac{differenz * 10^6}{CLOCKS_PER_SEC * count} [ns] \quad (22)$$

Abbildung 7 auf der vorherigen Seite zeigt den entsprechenden Code-Abschnitt der Zeitmessung.

6.1 systematische Fehler

Wie in Abbildung 7 auf der vorherigen Seite zu sehen ist, sind zwischen den beiden `clock()`-Aufrufen noch zusätzliche Befehle vorhanden, die mit in die Zeitmessung einfließen aber nicht mit der zu testenden Funktion in Bezug stehen. Dieser systematische Fehler ist bei allen Funktionen gleich groß. Aber auch die Register-Zuweisungen der `__asm__()`-Blöcke werden durch die Messung erfasst. Dieser Fehler ist für die jeweiligen Funktionen allerdings unterschiedlich groß und muss daher auch für jede Funktion ermittelt werden. Somit besitzt jede zu testende Funktion ihren eigenen Fehler, den Offset.

Um den Offset zu bestimmen, habe ich in den zu testenden Funktionen die Assembler Befehle entfernt und mit den leeren `__asm__()`-Blöcken eine Laufzeitmessung durchgeführt. Somit sind alle Einflüsse, die nicht durch die Assembler Befehle verursacht werden, bestimmt und können bei den eigentlichen Messungen subtrahiert werden.

7 Anwendung Benchmark

Der Benchmark läuft bisher nur unter einem Linux Betriebssystem und benötigt neben dem Softwarepaket *GMP* noch die Softwarekomponente "libpolynom". Um den Benchmark zu starten, müssen diese per Umgebungsvariable `LD_LIBRARY_PATH` auffindbar

sein. Diese Aufgabe übernimmt ein Shell-Skript. Das Skript befindet sich im Verzeichnis `"/Benchmark/bin"` auf der beiliegenden CD. Auf der CD befindet sich neben der Softwarekomponente `"libpolynom"` (`/Benchmark/bin/lib`) auch die Sourcen des Benchmarks (`/Benchmark/src`).

Wenn man den Benchmark mit der Parameter `"-h"` aufruft, erhält man eine Hilfe, die die möglichen Parameter erklären. Diese Hilfe ist in Abbildung 8 zu sehen.

```
Aufruf: Benchmark [Optionen]

Optionen:
-----

-p=[ Ad] ,[ A(d-1)] ,...,[ A0] Polynom mit Grad d:
                               A(d-i) kennzeichnen die einzelnen Koeffizienten
                               Es koennen mehrere Polynome mittels -p angegeben
                               werden

-t=Zeit [ 1]      *           Zeitintervall, in der der Test laufen soll [ s]
-X=Wert [ 5]      *           Eingabe fuer X
-x=Wert [ 3]      *           Eingabe fuer x
-c               *           Bestimmt, ob x waehrend des Tests konstant sein soll

-A               *           Alle Algorithmen testen
-B               *           Bshouty testen (Inline-Assembler)
-H               *           Horner testen (Inline-Assembler)
-L               *           Look-Up-Table testen (Inline-Assembler)

-h               *           Hilfe

[*] : Standard-Optionen
```

Abbildung 8: Parameter des Benchmarks

Nachdem man die Umgebungsvariable `LD_LIBRARY_PATH` entsprechend angepasst hat, kann man den Benchmark starten. Die Ausgabe von `"/Benchmark -X=3 -x=2 -t=4 -A"` ist in Abbildung 9 dargestellt.

```

=====
= RAM-Modell und 64-Bit Prozessoren =
=====

Benchmark compiliert am Apr  9 2008 um 21:47:37

Systeminformationen:
-----

Di 15. Apr 00:01:52 CEST 2008

Linux Sphere 2.6.24.3 #1 SMP Tue Feb 26 20:09:17 CET 2008 x86_64 x86_64 x86_64
GNU/Linux

 00:01:52 up  8:54,  1 user,  load average: 0,05, 0,08, 0,03
USER      TTY      LOGIN@  IDLE   JCPU   PCPU WHAT
alex      :0        15:08  ?xdm?  10:50  0.02s /bin/sh /opt/kde3/bin/startkde

Gegebene Polynome:
-----

1*x^4 + 0*x^3 + 0*x^2 + 3*x^1 + 2

Definitionsbereich:
-----

x: 3
X: 3

Benchmark Optionen:
-----

Zeitintervall:          4 s
x konstant:             nein
Algorithmen:           Bshouty, ASM Horner, Look-Up Table,

=====
***** Benchmark laeuft *****
=====

Polynom 1 von 1
Algorithmus; Polynom; Grad; x; X; Ergebnis; Z; p(Z); Iterationen; Gesamtlaufzeit [ ms];
Einzellaufzeit [ ns]
Bshouty; 1*x^4 + 0*x^3 + 0*x^2 + 3*x^1 + 2; 4; 0; 3; 2; 512; 68719478274; 103701000;
3990; 38.4760
ASM Horner; 1*x^4 + 0*x^3 + 0*x^2 + 3*x^1 + 2; 4; 2; 3; 24; -; -; 390956000; 3990;
10.2058
Look-Up Table; 1*x^4 + 0*x^3 + 0*x^2 + 3*x^1 + 2; 4; 0; 3; 2; -; -; 1060070000; 4000;
3.7733

```

Abbildung 9: Ausgabe des Aufrufs `./Benchmark -X=3 -x=2 -t=4 -A`

Die hier ermittelten Messwerten sind noch alle Offset behaftet. Der Offset muss noch von den Messwerten subtrahiert werden.

8 Testumgebung

Die Tabelle 4 beschreibt die Gegebenheiten der Plattformen, auf denen die Tests laufen. Beide Betriebssysteme sind mit Unterstützung für den so genannten *Compatibility Mode* übersetzt worden. Damit ist es möglich, nicht nur 64Bit Programme auszuführen, sondern auch 32Bit Programme. Daher werden die Tests sowohl mit 32Bit Genauigkeit als auch mit 64Bit Genauigkeit durchgeführt, im folgenden 32Bit Mode bzw. 64Bit Mode genannt.

	AMD	Intel
Prozessor:	Athlon64 X2 4800+	Core 2 Quad Q6600
L1 Cache:	2x 128KB	4x 32KB
L2 Cache:	2x 1024 KB	4x 4096 KB
Core:	2	4
Takt:	2,4GHz	2,4GHz
RAM:	2GB	4GB
Betriebssystem:	Linux From Scratch 6.3	OpenSuse 10.3
Kernel:	2.6.24.3	2.6.22.5-31-default
GCC:	4.2.3	4.1.2

Tabelle 4: Testumgebung

9 Testdurchführung

Dieser Abschnitt beschreibt die einzelnen Tests, die die in Abschnitt 2 auf Seite 6 aufgestellten Fragen beantworten. Die jeweilige Messung dauert 10 Sekunden und wird 10-mal wiederholt.

1. Bestimmung des Offset für die jeweilige Messung
2. Die Messung der Laufzeit der Algorithmen erfolgt mit dem Polynom $P = 4x^3 + 3x^2 + 2x + 1$. Der Definitionsbereich ist mit $X = 3$ festgelegt.
3. Die Messung der Laufzeit der Einzeloperation von Bshoutys Algorithmus erfolgt ebenfalls mit dem Polynom $P = 4x^3 + 3x^2 + 2x + 1$. Der Definitionsbereich ist wiederum mit $X = 3$ festgelegt.
4. Die Messung der Laufzeit bei steigendem Grad d erfolgt mit dem Polynom $P = \sum_{i=0}^d x^i$. Der Definitionsbereich ist mit $X = 3$ festgelegt.
5. Die Messungen bei den Punkten 4) und 6) erfolgen mit variabel als auch mit konstanter Auswertestelle x .
6. Der Einfluss des Caches wird durch Setzen des Definitionsbereich X bestimmt. Der Definitionsbereich ist mit $X = 10^7$ festgelegt. Die Auswertung erfolgt an dem Polynom $P = 4x^3 + 3x^2 + 2x + 1$.
7. Jede Messung 1) bis 5) wird im 32Bit Mode als auch im 64Bit Mode durchgeführt.

10 Auswertungen

Im folgenden finden die Auswertungen der verschiedenen Messungen statt. Um nicht jedes Mal die Angaben für die Testumgebung zu wiederholen, werden folgende Abkürzungen vorgenommen:

Kürzel	Bedeutung
BSHOUTY	Bshoutys Algorithmus
HORNER	Horner-Schema
BSHOUTY32	Bshoutys Algorithmus im 32Bit Mode
BSHOUTY64	Bshoutys Algorithmus im 64Bit Mode
HORNER32	Horner-Schema im 32Bit Mode
HORNER64	Horner-Schema im 64Bit Mode
BSHOUTY_AMD32	Bshoutys Algorithmus im 32Bit Mode auf der AMD Plattform
BSHOUTY_AMD64	Bshoutys Algorithmus im 64Bit Mode auf der AMD Plattform
BSHOUTY_INTEL32	Bshoutys Algorithmus im 32Bit Mode auf der Intel Plattform
BSHOUTY_INTEL64	Bshoutys Algorithmus im 64Bit Mode auf der Intel Plattform
HORNER_AMD32	Horner-Schema im 32Bit Mode auf der AMD Plattform
HORNER_AMD64	Horner-Schema im 64Bit Mode auf der AMD Plattform
HORNER_INTEL32	Horner-Schema im 32Bit Mode auf der Intel Plattform
HORNER_INTEL64	Horner-Schema im 64Bit Mode auf der Intel Plattform

Tabelle 5: Abkürzungen der Testumgebungen

10.1 Laufzeit bei einem fest vorgegebenen Polynom

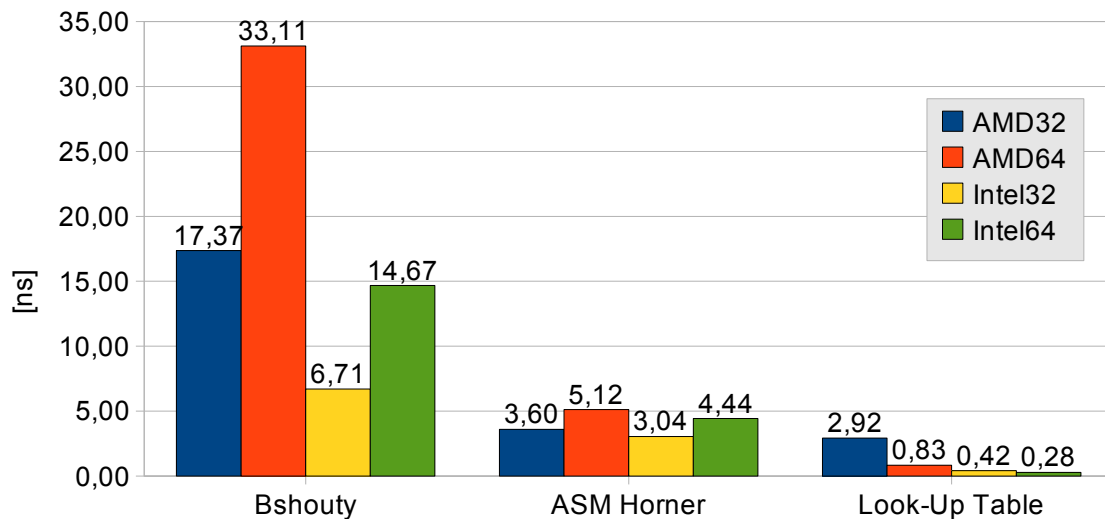


Abbildung 10: Laufzeit bei einem fest vorgegebenen Polynom

In Abbildung 10 sind die Laufzeiten der Algorithmen unter den jeweiligen Umgebungen zu sehen. Es fällt hier auf, dass die Laufzeiten für Bshoutys Algorithmus mit keinem anderen Verfahren konkurrieren kann. Selbst mit dem besten Wert, den BSHOUTY_INTEL32 erzielt (6,71ns), ist BSHOUTY langsamer als HORNER_AMD64 (5,12ns), das die schlechteste Laufzeit besitzt. Hier ist BSHOUTY im Vergleich zu den anderen Verfahren die mit Abstand schlechteste Möglichkeit, ein Polynom auszuwerten. Dieses Ergebnis war aber auch zu erwarten: Erst bei Grad $d \geq 5$ für die 32Bit Umgebung bzw. $d \geq 7$ für die 64Bit Umgebung sollte BSHOUTY schneller als das Horner-Schema sein. Doch da der Grad des fest vorgegebenen Polynoms nur 3 betrug, konnte Bshoutys Algorithmus somit auch nicht performanter sein.

Die Look-up Tabellen erzielen die besten Laufzeiten. Eine Ausnahme stellt der Wert unter dem AMD32 System dar. Ob es sich um einen Messfehler handelt oder nicht, kann nicht ermittelt werden.

Vergleicht man die theoretische Laufzeit unter der AMD Umgebung, so stellt man fest, dass die theoretischen Werte mit den gemessenen gut übereinstimmen. Nur im 32Bit Mode ist die gemessene Laufzeit weniger als die theoretische:

AMD32	Wert	Abweichung vom theoretischen Wert
Theoretischer Wert:	19,58ns	-
Messwert:	17,37ns	-11,3%

AMD64	Wert	Abweichung vom theoretischen Wert
Theoretischer Wert:	33,75ns	-
Messwert:	33,11ns	-1,9%

Tabelle 6: Vergleich Messwerte - Theoretische Werte

10.2 Laufzeit der Einzeloperationen bei BSHOUTY

Abbildung 11 zeigt die Laufzeit der jeweiligen Operationen von BSHOUTY. Mit einem Blick wird deutlich, dass der Divisionsbefehl **div** limitierende Faktor ist, der BSHOUTY ausbremst. Kein anderer Befehl ist so langsam wie die Division. Die theoretischen Laufzeiten werden hier bestätigt.

AMD32	Wert	Abweichung vom theoretischen Wert
Theoretischer Wert:	16,25ns	-
Messwert:	16,10ns	-0,9%

AMD64	Wert	Abweichung vom theoretischen Wert
Theoretischer Wert:	29,58ns	-
Messwert:	31,43ns	+6,2%

Tabelle 7: Vergleich Messwerte - Theoretische Werte

Bemerkenswert ist auch, dass die Laufzeit einiger Operationen negativ ist. Dies ist durch

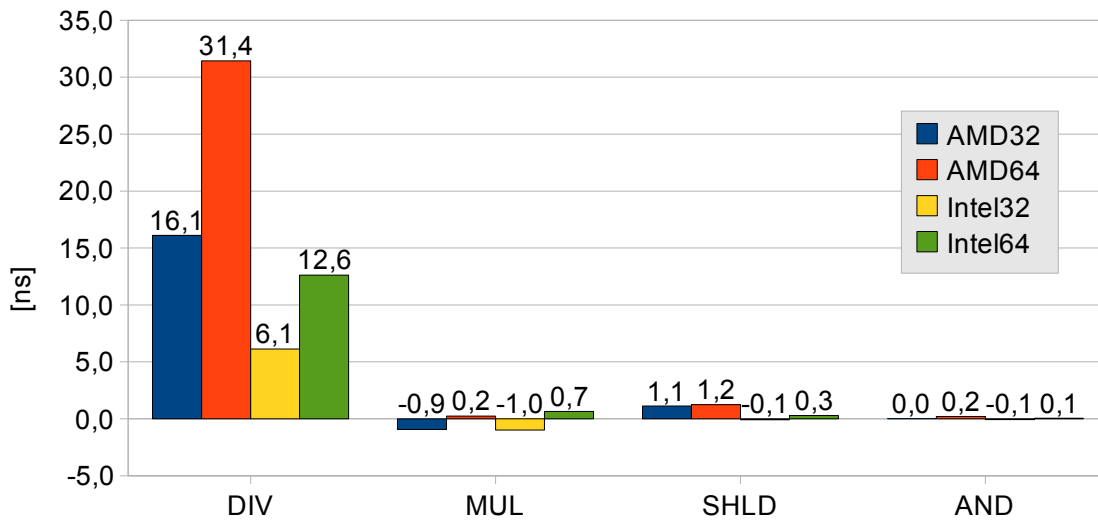


Abbildung 11: Laufzeit der Einzeloperationen bei Bshoutys Algorithmus - Auswertestelle x variabel

Messtoleranzen verursacht worden. Beim *MUL* Befehl ist die negative Laufzeit erklärbar, die im 32Bit Mode ermittelt wurden:

Ein Blick auf die Implementierung von BSHOUTY auf Seite 14 macht deutlich, was geschieht. Um den *MUL* Befehl zu testen, sind alle anderen Befehle aus dem `__asm__()` Block entfernt worden, so dass nur noch der *MUL* Befehl vorhanden ist. Auf diese Weise wird der Registerinhalt `%eax` mit `%edi` multipliziert! Unglücklicherweise werden im Register `%eax` die unteren 32 Bits von Z^{d+1} gespeichert.

Es gilt bei diesem Polynom: $Z = 512 = 2^9$. Da $Z^{d+1} = 2^{9*(3+1)} = 2^{36}$ ist, sind alle Bits im Register `%eax` unbesetzt, das heißt: das Register `%eax` ist komplett mit 0 gefüllt und nur im Register `%edx` ist ein einziges Bit gesetzt. Bei der Multiplikation ist also ein Faktor 0, sodass die Multiplikation nicht durchgeführt werden muss. Das Ergebnis ist 0 unabhängig vom zweiten Faktor - $P(Z)$. Das scheint der Prozessor zu erkennen und muss hier Optimierungen vornehmen, die dazu führen, dass der vom Offset bereinigte Messwert negativ wird.

10.3 Laufzeit bei steigendem Grad eines Polynoms

Umgebung 64Bit

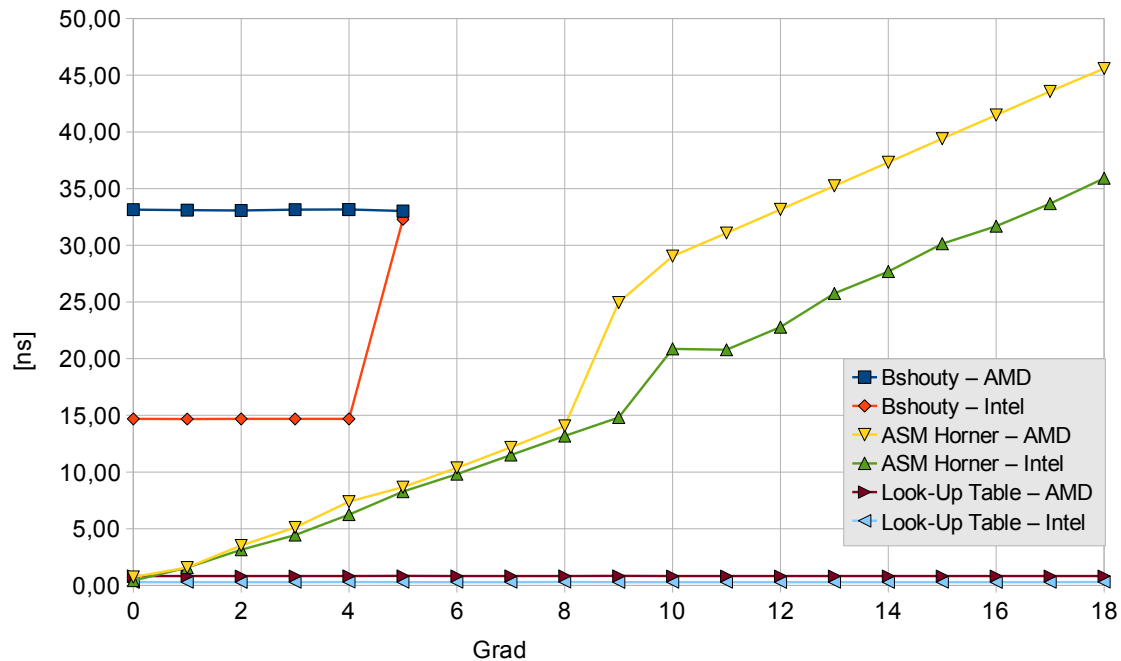


Abbildung 12: Laufzeit bei steigendem Grad eines Polynoms - Systemumgebung: 64Bit

In Abbildung 12 erkennt man, dass die Look-up Tabellen wie erwartet nicht vom Grad des Polynoms abhängen. Die Laufzeiten sind konstant. Allerdings ist die Intel64 Plattform leicht im Vorteil gegenüber der AMD64 Plattform.

Wie erwartet sind die Laufzeiten von HORNER64 nicht konstant, sondern hängen linear vom Grad des Polynoms ab. Eine Besonderheit stellt die AMD64 Plattform dar. Bei Grad 9 gibt es einen sprunghaften Anstieg der Laufzeit. Vermutlich werden hier Cache Grenzen überschritten. Aber genau erklärbar ist der Anstieg nicht. Bei der Intel64 Plattform sieht man bei Grad 10 auch eine leichte Abweichung, die sich aber nach dem Grad 11 wieder normalisiert. Daher wird die Funktion als linear betrachtet. Bei der AMD64 Plattform wird zwischen Grad $d < 8$ und Grad $d > 8$ unterschieden. Somit gibt es 3 Funktionen für beiden Plattformen. Sie lauten:

	AMD64	Intel64
Grad $d < 9$	$f_1 = a_1 + xb_1$	$f_3 = a_3 + xb_3$
Grad $d \geq 9$	$f_2 = a_2 + xb_2$	$f_3 = a_3 + xb_3$

Tabelle 8: Gradgleichungen HORNER64

Mittels numerischer Regression lassen sich die Koeffizienten der Funktionen bestimmen - (der Grad ist einheitenlos):

AMD64				
$f_i = a_i + xb_i$	a_i	Fehler σ_a	b_i	Fehler σ_b
Grad $d < 9$	0,24ns	0,16ns	1,71ns	0,03ns
Grad $d \geq 9$	8,28ns	0,03ns	2,07ns	0,002ns

Intel64				
$f_i = a_i + xb_i$	a_i	Fehler σ_a	b_i	Fehler σ_b
Grad d	-1,53ns	0,47ns	2,06ns	0,044ns

Tabelle 9: Koeffizienten der Gradgleichungen bei HORNER64

Nach Grad 9 sind die Steigungen der Gradgleichungen nahe zu identisch. Dies wird auch aus dem Diagramm ersichtlich. Beide Kurven laufen fast parallel zueinander. Allerdings unterscheiden sich die beiden Funktionen in ihrem y-Achsenabschnitt. Hier besitzt HORNER_AMD64 einen größeren y-Achsenabschnitt.

BSHOUTY_AMD64 ist konstant und beträgt, wie in Abbildung 10 auf Seite 23 zu sehen ist, bei $(33, 11 \pm 0, 05) ns$. Dagegen verharrt BSHOUTY_INTEL64 bis zum Grad 4 bei dem Wert, der auch in Abbildung 10 auf Seite 23 zu sehen ist: $14, 67 \pm 0, 01 ns$. Bei Grad 5 hingegen gibt es einen großen Sprung in der Laufzeit. Die Laufzeit steigt um mehr als das Doppelte an. Sie gleicht sich der Laufzeit der AMD64 Plattform an. BSHOUTY_INTEL64 ist also nicht konstant!

Weiterhin erkennt man den Punkt, ab dem BSHOUTY_AMD64 schneller ist als HORNER: bei der AMD64 Plattform ist das Grad 13 und bei der Intel64 Plattform sogar erst bei Grad 18. Die aus den Messungen gewonnene Steigung für HORNER ist um ein vielfaches kleiner als die Steigung, die aus den theoretischen Vorüberlegungen ermittelt wurden. Daher musste auch der Grad steigen, ab dem BSHOUTY schneller ist als HORNER. BSHOUTY_INTEL64 ist im besten Fall ab Grad 8 auf der AMD64 Plattform bzw. bei Grad 9 auf der Intel64 Plattform performanter als HORNER.

32Bit Umgebungen

In Abbildung 13 sind die Laufzeiten für die 32Bit Umgebungen zu sehen. Dort kann man erkennen, dass die Look-up Tabellen wiederum konstante Laufzeiten aufweisen. Allerdings ist die AMD32 Plattform stark im Nachteil gegenüber der Intel32 Plattform.

Wie im 64Bit Mode ist HORNER32 auch hier vom Grad des Polynoms abhängig. Bei HORNER_INTEL32 gibt es zum Teil erhebliche Abweichungen. Da die Abweichungen aber nur zwischen Grad 8 und 13 liegen, gehe ich hier wiederum von einem linearen Anstieg aus. Die Koeffizienten der Geradengleichungen werden wieder mittels numerischer Regression bestimmt:

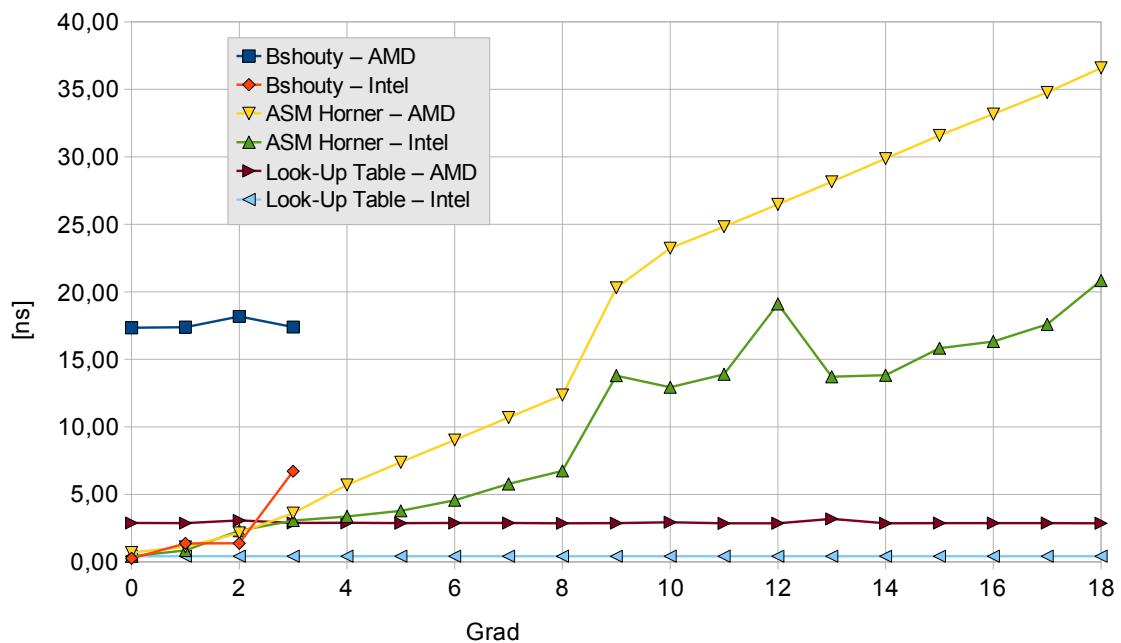


Abbildung 13: Laufzeit bei steigendem Grad eines Polynoms - Systemumgebung: 32Bit

AMD32				
$f_i = a_i + xb_i$	a_i	Fehler σ_a	b_i	Fehler σ_b
Grad $d < 9$	$-0,34ns$	$0,30ns$	$1,55ns$	$0,06ns$
Grad $d \geq 9$	$6,50ns$	$0,09ns$	$1,67ns$	$0,007ns$

Intel32				
$f_i = a_i + xb_i$	a_i	Fehler σ_a	b_i	Fehler σ_b
Grad d	$-0,34ns$	$0,90ns$	$1,14ns$	$0,09ns$

Tabelle 10: Koeffizienten der Gradgleichungen bei HORNER32

BSHOUTY_AMD32 ist konstant und liegt sogar auf einem kleineren Level als BSHOUTY_AMD64. Dafür ist aber auch bei Grad 3 die Laufzeitmessung zu Ende, da mit $P(Z)$ offensichtlich die 32Bit Grenze überschritten wird. Interessant ist der Kurvenverlauf von BSHOUTY_INTEL32. Dieser ist in Abbildung 14 nochmals vergrößert dargestellt.

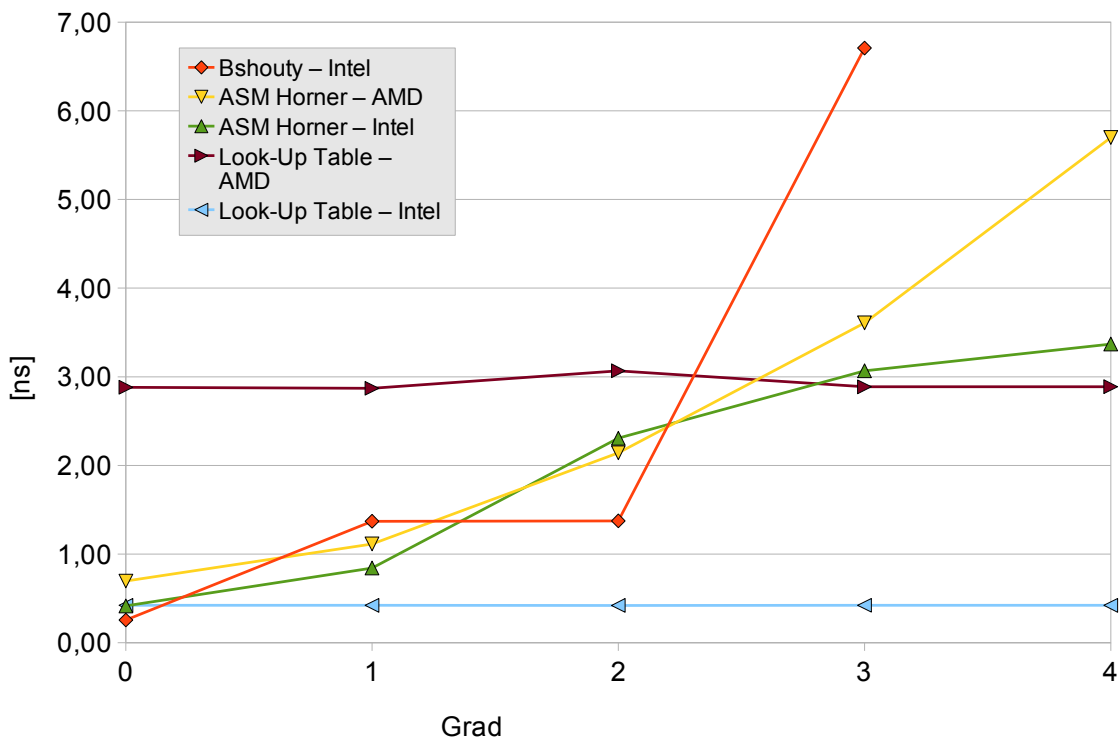


Abbildung 14: Laufzeit bei steigendem Grad eines Polynoms - Vergrößerung - Systemumgebung: 32Bit

Dort kann man erkennen, dass die Laufzeit von BSHOUTY_INTEL32 mit dem Grad d ansteigt. Bei Grad $d = 0$ ist BSHOUTY_INTEL32 der schnellste Algorithmus und unterbietet damit sogar die Laufzeit der Look-up Tabellen: Die Auswertung einer Konstante ist mit BSHOUTY_INTEL32 schneller möglich als ein Zugriff auf den Hauptspeicher! Sogar HORNER_INTEL32 ist hier langsamer. Eine mögliche Erklärung ist, dass nach der Zuweisung der Konstante p_0 in das Register `%edi` (Zeile 6 bei HORNER) der Test (Zeile 7 und 8), ob d größer oder gleich 1 ist, die Laufzeit vergrößert. Bei Grad $d = 2$ ist BSHOUTY_INTEL32 schneller als HORNER32. Wie BSHOUTY_INTEL64 ist auch BSHOUTY_INTEL32 nicht konstant! Warum das so ist, untersucht der Abschnitt 10.3.1.

10.3.1 Laufzeit bei steigendem Definitionsbereichs X bei dem Polynom $P = x^2 + x + 1$

Um den sprunghaften Anstieg von BSHOUTY_INTEL zu untersuchen wird, wird abermals das Polynom ausgewertet, bei dem die Laufzeit von BSHOUTY_INTEL32 geringer ist als die Laufzeit von HORNER_INTEL32, konkret: $x^2 + x + 1$. Jedoch wird jetzt der Definitionsbereich X schrittweise erhöht. Bei den 32Bit Umgebungen steigt der Definitionsbereich X von 1 auf 32 und bei den 64Bit Umgebungen von 1 auf 300. Dabei wird die Laufzeit von BSHOUTY und den Assembler Befehlen gemessen. Zusätzlich wird noch Z^{d+1} berechnet und mit in die folgenden Diagramme eingetragen.

Plattform AMD

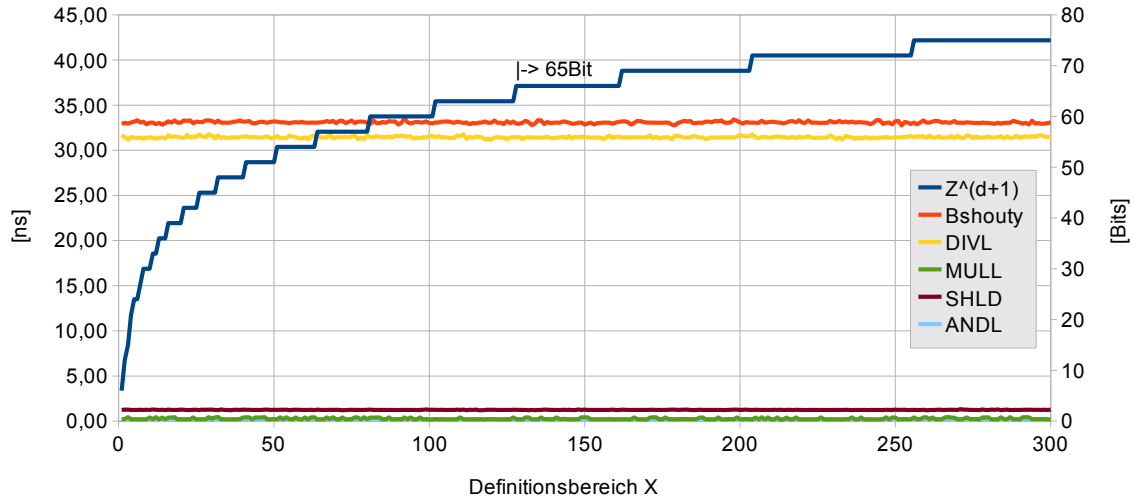


Abbildung 15: Laufzeit bei steigendem Definitionsbereichs X bei dem Polynom $P = x^2 + x + 1$ - Systemumgebung: AMD - 64Bit

In Abbildung 15 erkennt man, dass die Laufzeit von BSHOUTY_AMD und die Laufzeiten der Assembler Befehle konstant sind. Hier gibt es keinen sprunghaften Anstieg der Laufzeit. Das gleiche Bild zeigt sich auch in Abbildung 16. Dort sind ebenfalls die Laufzeiten konstant. Die Größe Z^{d+1} hat also offensichtlich überhaupt keinen Einfluss auf die Laufzeiten.

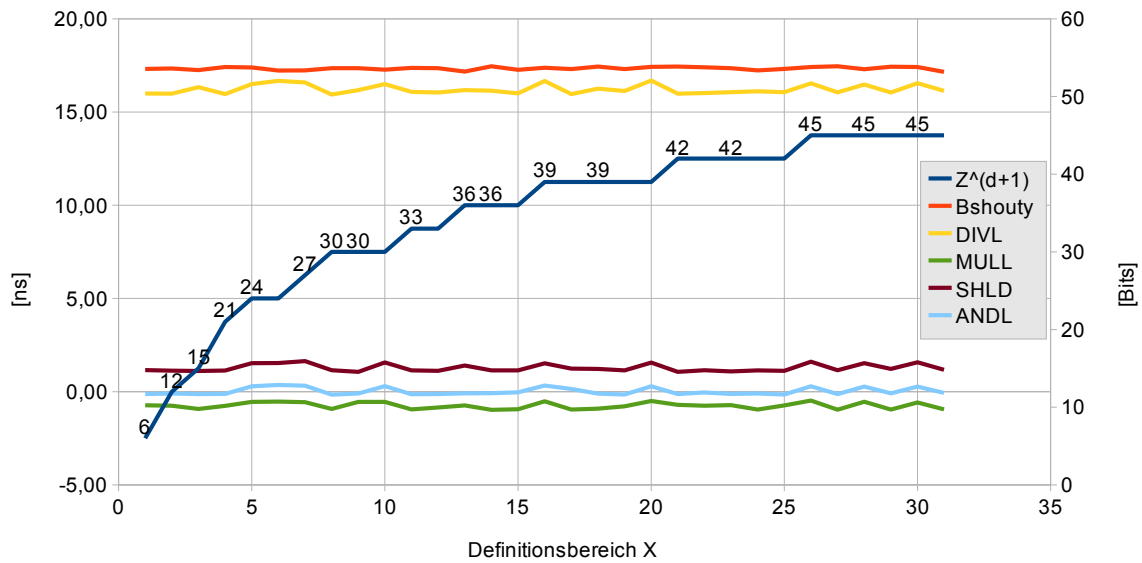
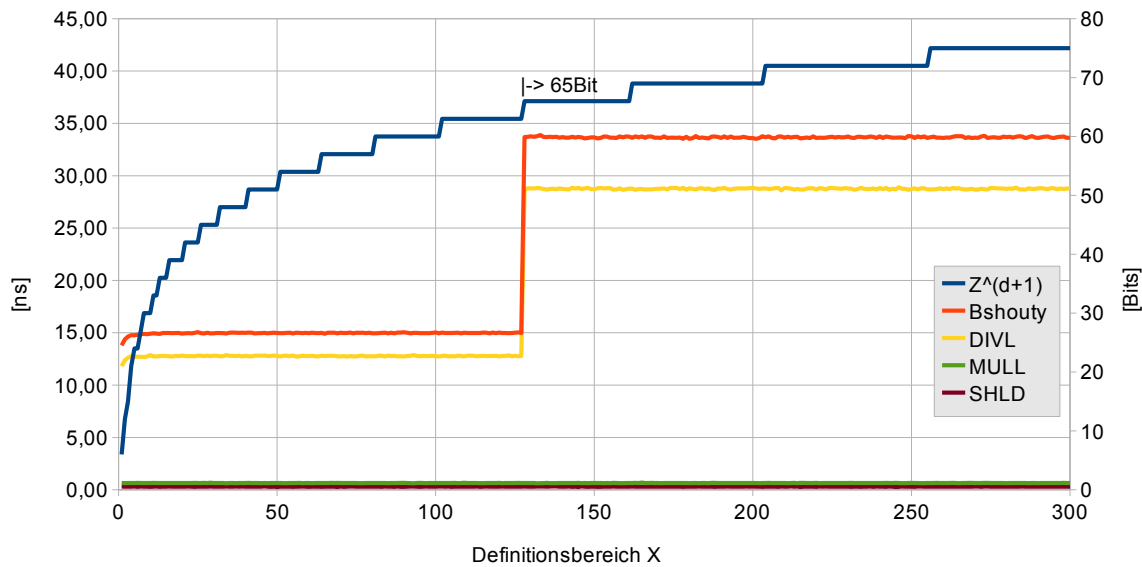


Abbildung 16: Laufzeit bei steigendem Definitionsbereichs X bei dem Polynom $P = x^2 + x + 1$ - Systemumgebung: AMD - 32Bit

Plattform Intel



liegen dagegen auf konstant niedrigem Niveau. Offensichtlich wird der Divisionsbefehl durch die Prozessor beschleunigt bzw. optimiert. Eine mögliche Erklärung ist:

Bei $Z^{d+1} < 2^{64}$ sind die oberen 64Bits unbesetzt. Das Register $\%rdx$ enthält also den Wert 0. So muss bei der Division nur der Wert im Register $\%rax$ durch den Wert im Register $\%rsi$ geteilt werden. Der Dividend ist maximal 64Bit groß. Bei $Z^{d+1} \geq 2^{64}$ ist Register $\%rax$ unbesetzt. Aber Register $\%rdx$ ist ungleich 0. Also ist der Dividend hier mindestens 64Bit groß, sodass die Division auch die beiden Register $\%rdx::\%rax$ berücksichtigen muss. Die Laufzeit der Division hängt also mit der Größe des Dividenden zusammen: Verdoppelt sich dieser, verdoppelt sich auch die Laufzeit der Division.

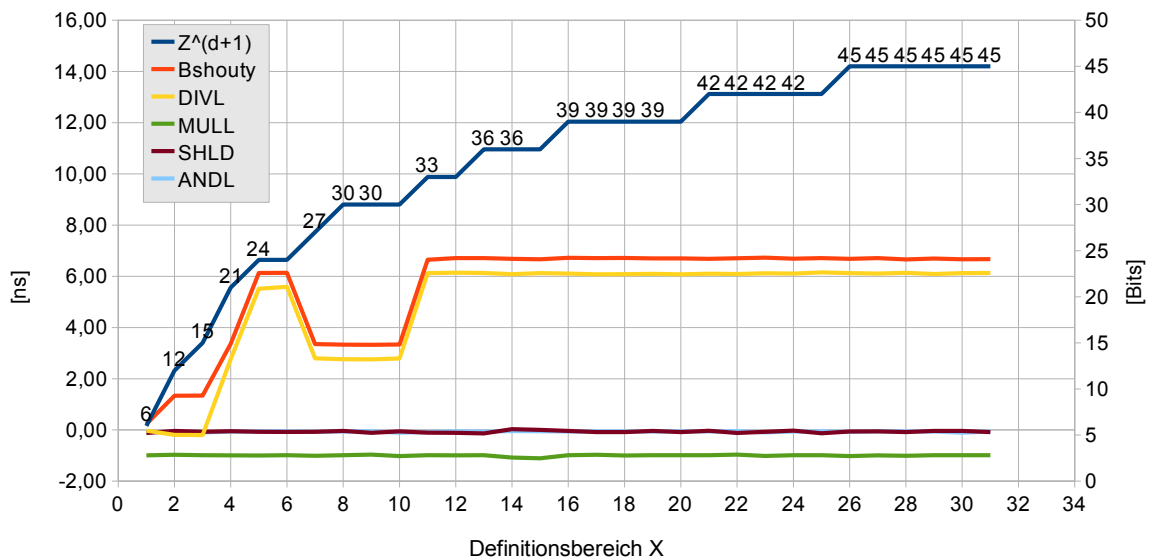


Abbildung 18: Laufzeit bei steigendem Definitionsbereichs X bei dem Polynom $P = x^2 + x + 1$ - Systemumgebung: Intel - 32Bit

Ähnlich sieht es in Abbildung 18 aus. Dort werden offensichtlich mehrere Optimierungen vorgenommen. Interessanterweise verlaufen die Kurven für BSHOUTY_INTEL32 und der Division annähernd parallel. Nur im Intervall $[1, 3]$ weicht die Laufzeit der Division von der Laufzeit BSHOUTY_INTEL32 ab. Vermutlich ist die Division mit einem maximal 16Bit großen Dividend sehr schnell durchzuführen. Im Intervall $[7, 10]$ sinkt sogar die Laufzeit von BSHOUTY_INTEL32. Welche Optimierungen hierfür verantwortlich sind, lässt sich nicht sagen.

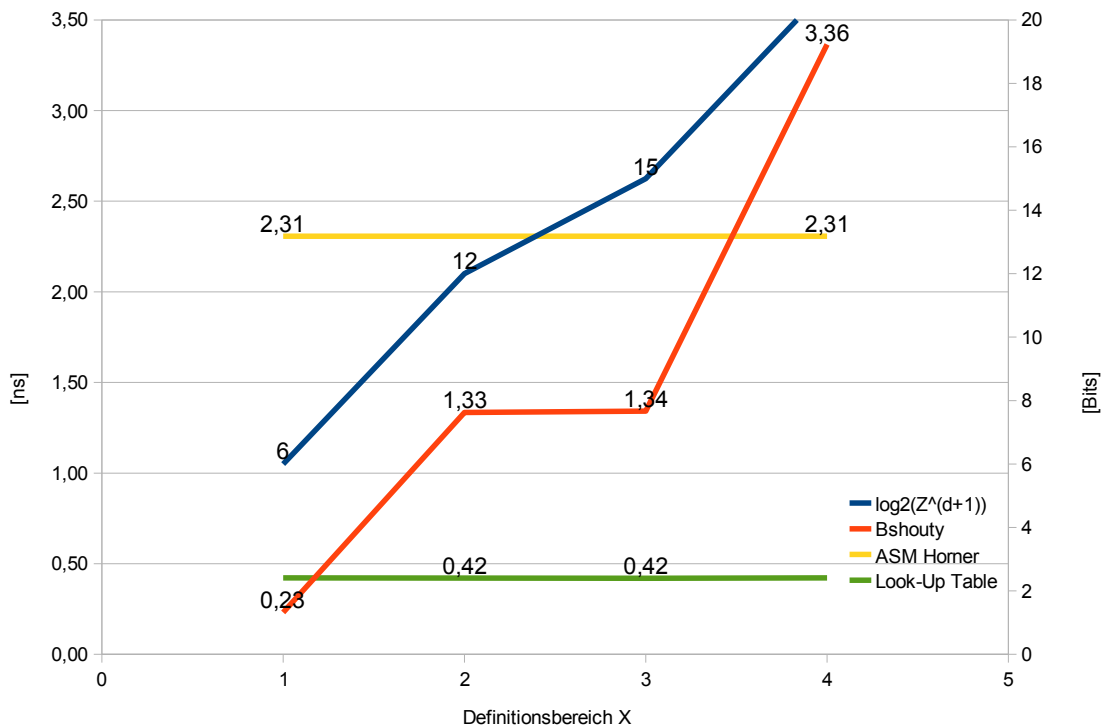


Abbildung 19: Laufzeit bei steigendem Definitionsbereichs X bei dem Polynom $P = x^2 + x + 1$ - Zoom - Systemumgebung: Intel - 32Bit

Zeichnet man jetzt HORNER_INTEL32 in das Diagramm und vergrößert den entsprechenden Bereich, erhält man Abbildung 19. Dort sieht man, dass der Definitionsbereich bis maximal $X = 3$ erweitert werden darf, damit BSHOUTY_INTEL32 schneller ist als HORNER_INTEL32. Ist $X \in \{1 \dots 3\}$, beträgt der Vorsprung knapp $1ns$. Bei einem Takt von $2,4GHz$ entspricht dies etwa $2,4Takte$. BSHOUTY_INTEL32 ist also bei Grad $d = 2$ und $X \in \{1 \dots 3\}$ performanter als HORNER. Es bleibt noch zu klären, bis zu welchem Wert von ρ BSHOUTY_INTEL32 schneller ist als HORNER. Das lässt aber sich ausrechnen. Aus den vorherigen Überlegungen ergibt sich, dass:

$$2^{16} \geq Z^{d+1}$$

Da $Z = 2^z$ und bei $d = 2$ folgt:

$$2^{16} \geq 2^{3z}$$

$$\Leftrightarrow 16 \geq 3z$$

$$\Leftrightarrow \frac{16}{3} \geq 3z$$

Da $t \in \mathbb{N}$ gilt:

$$5 \geq z$$

Nun kann man ρ berechnen:

$$Z = 2^z > \max \{X^d \rho, (X^d + 1)X\}$$

mit $d = 2$ und $X = 3$ folgt:

$$Z = 2^z > \max \{9\rho, 30\}$$

mit $t = 5$ gilt:

$$2^5 > 9\rho \Leftrightarrow 32 > 9\rho \Leftrightarrow \frac{32}{9} > \rho$$

Da $\rho \in \mathbb{N}$ gilt:

$$3 \geq \rho > 0$$

Falls $d = 2$, $X \in \{1 \dots 3\}$ und $\rho \in \{1 \dots 3\}$ ist BSHOUTY_INTEL32 schneller als HORNER.

10.4 Einfluss der Auswertestelle

Umgebung 64Bit

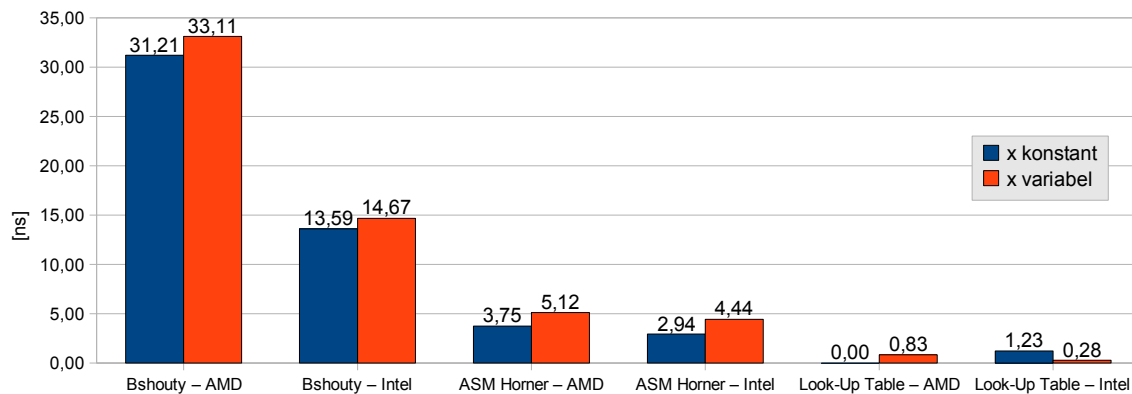


Abbildung 20: Einfluss der Auswertestellen - Systemumgebung: 64Bit

Aus Abbildung 20 wird ersichtlich, dass die wiederholte Auswertung eines Polynoms bei einem konstanten x schneller möglich ist als, wenn x bei den Auswertungen variiert. Nur die Laufzeit der Look-up Tabellen auf der Intel Plattform verhalten sich entgegen den Erwartungen. Eine Erklärung für dieses ungewöhnliche Verhalten lässt sich nicht finden. Bemerkenswert ist auch, dass, falls x konstant ist, die Laufzeit der Look-up Tabellen auf der AMD Plattform so gut wie gar keine Zeit beansprucht.

Umgebung 32Bit

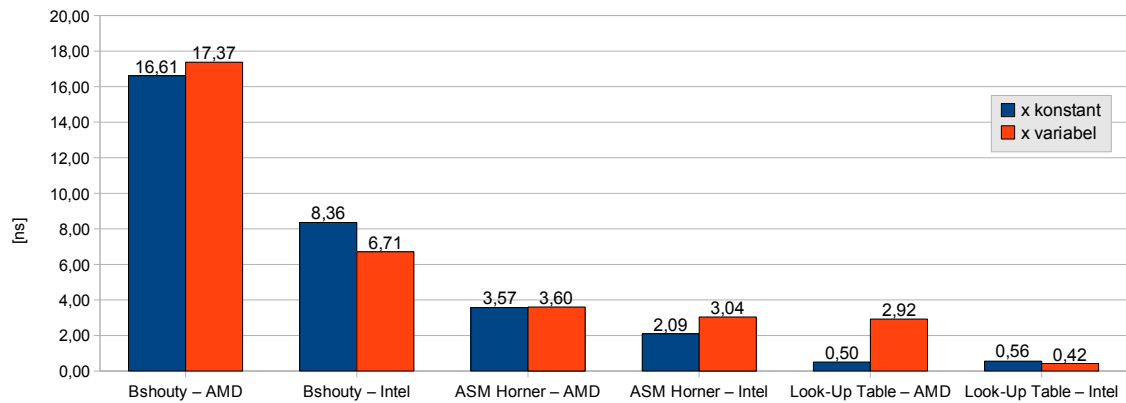


Abbildung 21: Einfluss der Auswertestellen - Systemumgebung: 32Bit

Eine Besonderheit zeigt Abbildung 21 bei der Laufzeit von BSHOUTY_INTEL. Hier sinkt sogar die Laufzeit, falls x bei der iterativen Auswertung variiert. Dies ist wieder eine Stelle, die sich nicht erklären lässt. Vermutlich hat die Anzahl der zu verarbeitenden Bits einen Einfluss auf die Laufzeit. Dagegen zeigt die Laufzeit von HORNER_AMD32 keinen signifikanten Unterschied in der Art der Auswertungen. Bei den Look-Up Tabellen explodiert förmlich die Laufzeit, falls x variiert. Hier steigt die Laufzeit unerklärbar um fast das 6-fache an.

Im Allgemeinen verhalten sich die Algorithmen dennoch wie erwartet: die Laufzeit steigt, falls x variiert. Ist x konstant, können etliche Werte aus dem Cache entnommen werden. Die Ausführungsgeschwindigkeit steigt an. Die Laufzeit sinkt.

10.5 Einfluss des Prozessor Cache

Da BSHOUTY bei dem Definitionsbereich $X = 10^7$ die 32Bit bzw. 64Bit Rechengenauigkeit bei weitem überschreitet, muss BSHOUTY bei diesem Test entfallen. Ein Blick auf die folgenden Abbildungen zeigt, dass HORNER unabhängig von der Größe von x ist. Daher findet nur die Auswertungen für die Look-up Tabellen statt.

Auswertestelle x konstant

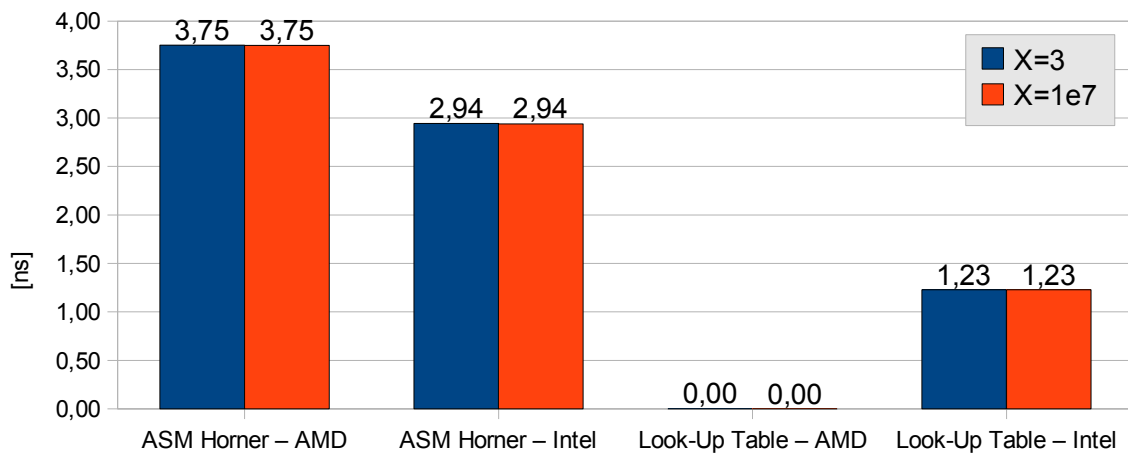


Abbildung 22: Einfluss des Prozessor-Cache - Auswertestelle x konstant - Systemumgebung: 64Bit

Auch beim zweiten Blick auf Abbildung 22 sieht man keine Unterschiede in den Laufzeiten. Die Messwerte sind bis auf die letzte Stelle gleich. Noch nicht einmal Messtoleranzen lassen sich hier ablesen. Wie schon in Abschnitt 10.4 gesehen, benötigt die Auswertung mit den Look-Up Tabellen (AMD64) keine messbare Zeit.

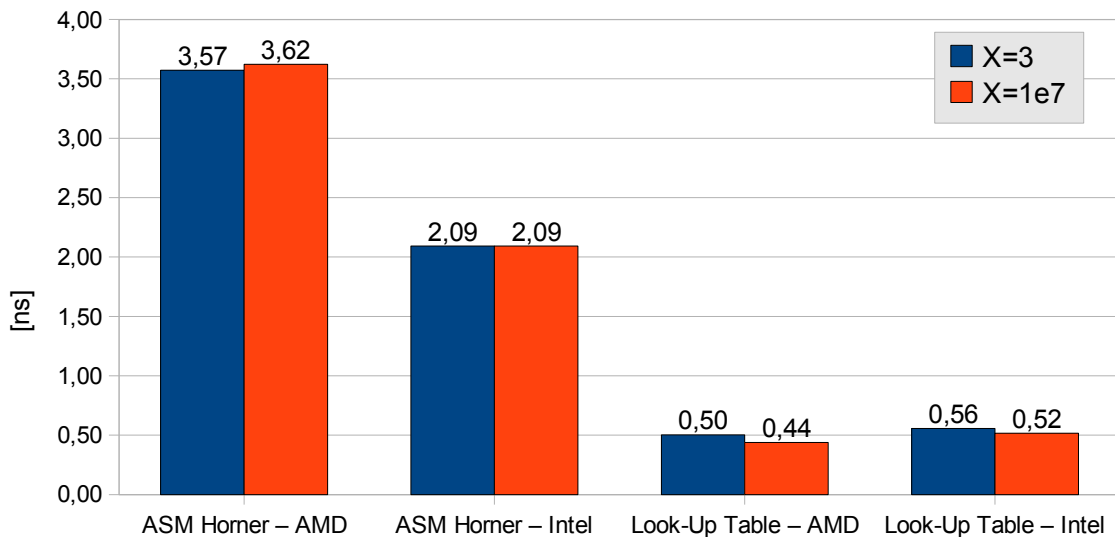


Abbildung 23: Einfluss des Prozessor-Cache - Auswertestelle x konstant - Systemumgebung: 32Bit

Etwas anders sieht es in Abbildung 23 aus. Hier schwanken die Laufzeiten doch deutlicher.

Die größte Abweichung ist bei den Look-up Tabellen (AMD32) mit -12% zu sehen. Dicht gefolgt von den Look-Up Tabellen (Intel) mit 7,1%.

Wenn bei der wiederholten Auswertung das Polynom an nur einer Stelle ausgewertet wird, macht es bei der 64Bit Umgebung keinen Unterschied, wie groß x ist. Die Messwerte bleiben dieselben. Dagegen schwanken die Messwerte bei 32Bit Umgebungen doch deutlich.

Auswertestelle x variabel

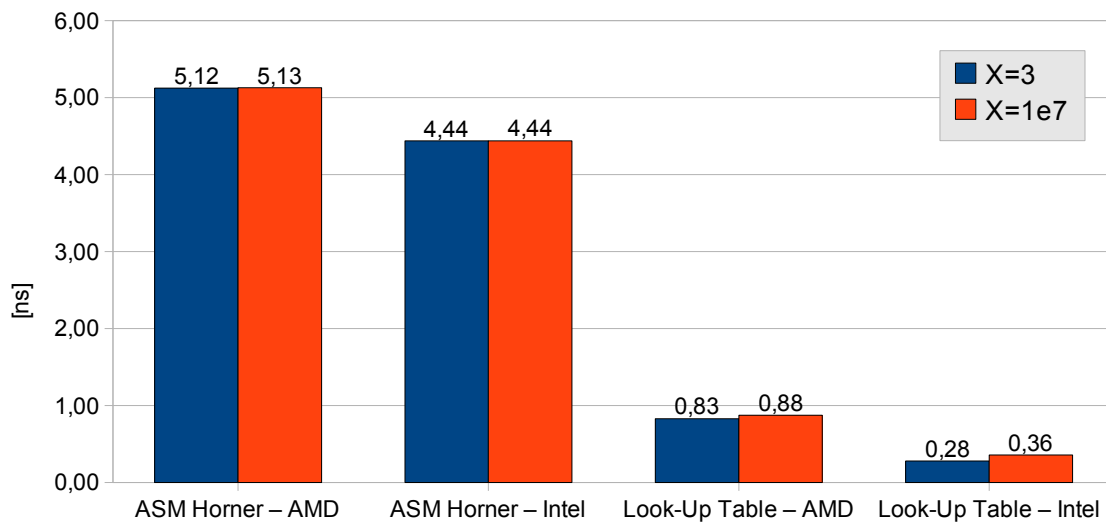


Abbildung 24: Einfluss des Prozessor-Cache - Auswertestelle x variabel - Systemumgebung: 64Bit

Bei Betrachtung der Abbildung 24 sieht man nicht sofort, dass die Größe von x unterschiedlich ist. Bei den Look-Up Tabellen (AMD) ist die Auswertung mit einem großen Wert von x nur 0,05ns langsamer. Dies entspricht aber einen Performanceverlust von 6%. Noch drastischer verlieren die Look-Up (Intel). Ihr Verlust beträgt 28,6%.

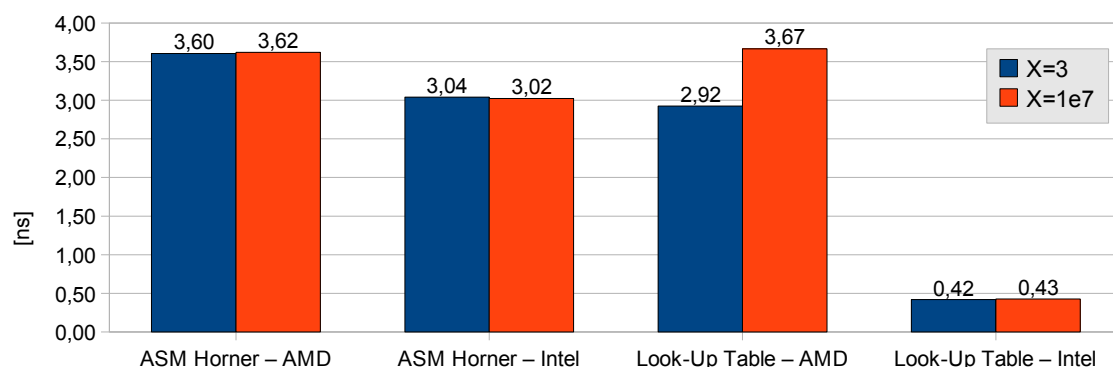


Abbildung 25: Einfluss des Prozessor-Cache - Auswertestelle x variabel - Systemumgebung: 32Bit

Ein völlig anderes Bild ist in Abbildung 25 zu sehen. Hier sind die Look-Up Tabellen (Intel) annähernd konstant, während auf der AMD Plattform die Laufzeit um 25,7% in die Höhe schnellt.

Insgesamt zeigt sich hier ein gemischtes Bild: Einmal sind die Look-Up Tabellen von der Größe x nicht abhängig, obwohl x bei der Auswertung variiert, ein anderes Mal nicht.

10.6 Einfluss des Prozessor Architektur

Die bisher dargestellten Diagramme zeigen:

1. Die Algorithmen sind auf einer 64Bit Plattform langsamer als auf einer 32Bit Plattform.
2. Die Ausführungsgeschwindigkeit der Algorithmen ist auf der Intel Plattform höher als auf der AMD Plattform.

Punkt 1 ist verständlich, da im 64Bit Mode doppelt so viele Bits verarbeitet werden müssen als im 32Bit Mode. Der Vergleich zwischen der AMD und Intel Architektur ist etwas unfair: Der AMD Prozessor wurde schon im Mai 2005 auf den Markt gebracht, während die Intel CPU erst im Januar 2007 auf den Markt kam. Daher ist auch verständlich, warum der Intel Prozessor diesen Vergleich für sich gewinnt.

11 Fehlerbetrachtung

Jede Messung ist fehlerbehaftet. Dieser Abschnitt beschreibt, wie man den mittleren Fehler der Einzelmessung, die Standardabweichung bestimmt und gibt Auskunft über die Messgenauigkeit der Zeitmessungen. Die lineare Regression ist aus [FR] übernommen. Allerdings habe ich mit N statt $N - 1$ bzw. $N - 2$ gerechnet, da mit $N - 1$ und $N - 2$ der Fehler überproportional ansteigt.

11.1 Standardabweichung

Wie in Abschnitt 9 auf Seite 22 wurden die Messungen $N = 10$ wiederholt. Somit kann man den mittleren Fehler σ bestimmen. Zunächst berechnet man das arithmetische Mittel aus den Messwerten:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad (23)$$

Die Varianz V berechnet sich mit:

$$V = \frac{1}{N} \sum_{i=1}^N (\bar{x} - y_i)^2 \quad (24)$$

Die Standardabweichung σ ergibt somit aus:

$$\sigma = \sqrt{V} \quad (25)$$

11.2 Lineare Regression

Mittels der linearen Regression lassen sich die Parameter a und b der Funktion $y = f(x) = a + bx$ so bestimmen, dass die Standardabweichung σ_y minimal wird. Die Standardabweichung σ_y berechnet sich dabei wie folgt:

$$\sigma_y^2 = \frac{1}{N} \sum_{i=1}^N (y_i - \bar{a} - \bar{b}x_i)^2 \quad (26)$$

\bar{a} und \bar{b} sind dabei die Mittelwerte der Parameter und berechnen sich wie folgt: Aus

$$\frac{\sigma_y^2}{\partial \bar{a}} = 0 \quad \text{und} \quad \frac{\sigma_y^2}{\partial \bar{b}} = 0 \quad (27)$$

folgt das Gleichungssystem:

$$\begin{aligned} \bar{a}N + \bar{b}[x] &= [y] \\ \bar{a}[x] + \bar{b}[x^2] &= [xy] \end{aligned} \quad (28)$$

mit:

$$[x] = \sum_{i=0}^N x_i \quad [y] = \sum_{i=0}^N y_i \quad [xy] = \sum_{i=0}^N x_i y_i \quad (29)$$

Das Gleichungssystem 28 wird gelöst durch:

$$\bar{a} = ([x^2][y] - [x][xy])/D \quad (30)$$

und

$$\bar{b} = (N[xy] - [x][y])/D \quad (31)$$

mit der Determinante des Gleichungssystems

$$D = N[x^2] - [x]^2 \quad (32)$$

Die Standardfehler $\sigma_{\bar{a}}$ und $\sigma_{\bar{b}}$ der Mittelwerte \bar{a} und \bar{b} werden wie folgt berechnet:

$$\sigma_y^2 = \frac{D}{[x^2]} \sigma_a^2 = \frac{D}{N} \sigma_b^2 \quad (33)$$

11.3 Fehlerabschätzung der Zeitmessung

Wie erwähnt erfolgt die Zeitmessung mittels der Funktion `clock()`. Dabei ist aufgefallen, dass die Zeitangaben $\pm 10\text{ ms}$ von den vorherbestimmten Wert abweicht. Daher wird der Fehler der Zeitmessung auf $\pm 10\text{ ms}$ geschätzt.

12 Fazit und Ausblick

Ist BSHOUTY praxisrelevant? So lautete die eingangs aufgeworfene Frage. Zweifelsohne ist BSHOUTY ein sehr pfiffige Methode, ein Polynom auszuwerten. Aber er leidet an der beschränkten Rechengenauigkeit des Prozessors. Auch ist die Division nicht gerade die schnellste Operation. Sie ist ausgesprochen langsam und bremst BSHOUTY aus. Immerhin kann BSHOUTY gegen HORNER einen kleinen Achtungserfolg erringen, wenn der Grad $d = 2$, $X \in \{1 \dots 3\}$ und $\rho \in \{1 \dots 3\}$ ist. Nur hier ist BSHOUTY schneller als HORNER. Gegen die Look-Up Tabellen kann sowohl HORNER als auch BSHOUTY nicht gewinnen. Der Speicherzugriff ist dafür einfach so schnell. Aber warum BSHOUTY_INTEL32 bei Grad $d = 0$ doch schneller als ein Speicherzugriff ist, kann nur vermuten werden: Vielleicht ist es doch nur ein Messfehler, der sich im Messverfahren oder in deren Implementierung versteckt. Jedenfalls bleibt der praxisnahe Einsatz von BSHOUTY auf den hier getesteten Prozessoren noch zweifelhaft.

In dieser Arbeit hat man auch gesehen, dass die Intel CPU bei der Division Optimierungen durchführt. Aber auch die aktuellen AMD Prozessoren, wie der Phenom [AMD07-1], führen Optimierungen am Divisionsbefehl durch. Hier wäre es noch interessant gewesen, die Laufzeiten der Algorithmen mit den aktuellen AMD Prozessoren zu testen. Aber warum sollte man denn auf Standard Prozessoren BSHOUTY laufen lassen? Vielleicht ist BSHOUTY auf für ihn abgestimmte Prozessoren tatsächlich schneller als HORNER und/oder als die Look-Up Tabellen auf Standard CPUs?

Literatur

- [LBZ07] K. Lürwer-Brüggemeier and M. Ziegler, "On Faster Integer Calculations using Non-Arithmetic Primitives", [HTTP://DE.ARXIV.ORG/ABS/0709.0624](http://DE.ARXIV.ORG/ABS/0709.0624)
- [AMD07] *AMD64 Architecture Programmer's Manual vol.: 1*: "Application Programming", Publication #24592 (Revision 3.14, September 2007)
- [AMD05] *Software Optimization Guide for AMD64 Processors*, Publication #25112 (Revision 3.06, September 2005)
- [AMD07-1] *Software Optimization Guide for AMD Family 10h Processors*, Publication #40546 (Revision 3.05, Dezember 2007)
- [AMD07-2] *AMD64 Architecture Programmer's Manual vol.: 3*: "General-Purpose and System Instructions", Publication #24594 (Revision 3.14, September 2007)
- [GCC] *GCC-Inline-Assembly-HOWTO*, v0.1, 01 March 2003, Sandeep.S, [HTTP://WWW.IBIBLIO.ORG/GFERG/LDP/GCC-INLINE-ASSEMBLY-HOWTO.HTML](http://WWW.IBIBLIO.ORG/GFERG/LDP/GCC-INLINE-ASSEMBLY-HOWTO.HTML)
- [FR] *Universität GH Paderborn - Fachbereich Physik - Physikalisches Praktikum - Fehlerrechnung*

Versicherung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe sowie ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Paderborn, den 15. April 2008

Alexander Spot