DIPLOMARBEIT

Geschwindigkeits-Optimierung eines Spracherkenners für Handheld PCs

von

Thilo Wolfgang Köhler

eingereicht am 26.11.2004
beim Institut für Logik, Komplexität
und Deduktionssysteme der Universität Karlsruhe (TH)

Referent: Prof. Dr.rer.nat. Alexander Waibel
Betreuer: Dipl.-Inform. Christian Fügen und
Dipl.-Inform. Sebastian Stüker

Ehrenwörtliche Erklärung

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 26.11.2004

Thilo Köhler

Kurzfassung

In dieser Arbeit wurde ein Spracherkenner mit Kenndaten, wie sie auf einem Desktop PC üblich sind, auf einen Handheld PC übertragen. Aufgrund der eingeschränkten Speicherkapazität und Rechenleistung sind neben der Portierung eine Vielzahl von Optimierungen unumgänglich. Diese Arbeit beschäftigt sich mit der Beschleunigung aller rechenintensiven Vorgänge von der Vorverarbeitung bis hin zur Decodierung. Viele der hier vorgestellten Optimierungen beruhen auf der Vermeidung von Fließkommazahlen, die auf einem heutigen Handheld PC nur emuliert und somit unter großen Geschwindigkeitseinbußen zur Verfügung stehen. Es werden aber auch Algorithmen vorgestellt, die eine Beschleunigung methodisch erzielen und daher auch auf anderen Systemen eingesetzt werden können.
Auf dem Test-PDA dieser Arbeit konnte die Vorverarbeitung um den Faktor 7 beschleunigt werden. Die Berechnung der Mahalanobis-Distanzen konnte mit einer Kombination verschiedener Optimierungen etwa um den Faktor 28 beschleunigt werden. Damit war es möglich, den Echtzeitfaktor auf dem 206MHz Prozessor des PDAs von über 52 auf 6,1 zu senken, ohne eine signifikante Verschlechterung der Wortakkuratheit in Kauf nehmen zu müssen. Durch Verengung des Suchstrahls der Decodierung konnte bei einem Verlust an Wortakkuratheit von weniger als 3% relativ ein Echtzeitfaktor von 2,9 erzielt werden. Eine weitere Verengung des Strahls brachte allerdings einen deutlichen Leitstungseinbruch mit sich.
Um tatsächliche Echtzeit bei gleichbleibender Wortakkuratheit zu erreichen, muss auf schnellere Hardware zurückgegriffen werden, oder es müssen weitere Optimierungen vorgenommen werden. Diese Optimierungen sollten eine Beschleunigung methodisch erzielen oder eine Reduzierung des Speicherverbrauchs bewirken, da der PDA an die Grenzen des Datendurchsatzes seines Arbeitsspeichers gestoßen ist.

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Die Anzahl der Menschen, die mobile Geräte wie Mobiltelefone oder PDAs[1] im täglichen Leben nutzen, nimmt ständig zu. Mit steigender Rechenleistung, Speicherkapazität und vielfältigeren Kommunikationsmöglichkeiten gewinnen diese Geräte mehr und mehr an Attraktivität. So können heute bereits viele verschiedene Aufgaben damit bewältigt werden, wie beispielsweise das Planen von Terminen, Schreiben von Texten oder Abspielen von Unterhaltungsmedien. Besteht eine Anbindung an das Internet, können auch E-Mails gesendet oder Inhalte von Internetseiten abgerufen werden.
Dabei muss die Steuerung mit einigen wenigen Knöpfen auskommen. Eine Tastatur, wie man sie bei einem Desktop PC findet, ist hier aufgrund der geringen Größe nicht möglich. Daher werden größere Handheld Systeme wie PDAs zusätzlich mit einem Stift auf einem berührungssensitiven Bildschirm bedient. Texte werden mit Hilfe einer kleinen, auf dem Bildschirm eingeblendeten Tastatur getippt oder können per Handschrifterkennung eingegeben werden. Diese Form der Eingabe ist aber oft nicht zufriedenstellend, etwa bei der Eingabe längerer Texte oder in Situationen, in denen nur eine Hand oder möglicherweise gar keine Hand frei ist. Als Beispiele hierfür seien die Steuerung eines Navigationssystems oder das Wählen einer Telefonnummer während der Autofahrt erwähnt.
Die Spracherkennung bietet sich hier als alternativer Eingabekanal an. Ein Mikrophon benötigt nur einige Millimeter physikalischen Platz und keinen haptischen Kontakt zum Anwender. Daher kann es auch robust an sehr kleinen Geräten angebracht werden. Aufgrund der eingeschränkten Leistung solcher Systeme war es aber in der Vergangenheit nur schwer möglich, mehr als einige wenige, sprecherabhängige Kommandos oder gar natürliche Sprache in Quasi-Echtzeit zu erkennen. Auch auf den heutigen Handheld PCs stößt man schnell an die Grenzen. Eine Reduzierung des Speicherverbrauchs und die Optimierung der Verarbeitungsgeschwindigkeit sind unumgänglich, möchte man eine vergleichbare Erkennungsleistung erreichen, wie sie bei Desktop PCs üblich ist.
Doch auch bei Erzielen von Quasi-Echtzeit der Erkennung ist die Aufgabe noch nicht gelöst, denn aus Sicht des Anwenders ist die Spracherkennung nur ein Eingabemedium und sollte die eigentliche Nutzanwendung nicht beeinträchtigen. Eine weitere wichtige Rolle spielt hier der Stromverbrauch, da mobile Geräte auf eine begrenzte Stromquelle angewiesen sind. Bei stärkerer Auslastung des Prozessors wird mehr elektrische Energie verbraucht als im Ruhezustand oder bei kurzzeitiger Belastung, und die mobile Nutzungsdauer des Gerätes sinkt.

[1] Ein PDA (*engl.* ***P**ersonal **D**igital **A**ssistant*) ist die übliche Bauform eines Handheld PCs.

1.2 Aufbau der Arbeit

Diese Arbeit beschäftigt sich hauptsächlich mit der Beschleunigung des Erkennungsvorgangs. Zu Beginn werden die allgemeinen Limitierungen von Handheld PCs näher betrachtet und die grundsätzliche Vorgehensweise der Optimierungen motiviert. Es werden Verweise auf verwandte Arbeiten gegeben, um gemeinsame Ansätze herauszustellen und Unterschiede zu anderen Arbeiten deutlich zu machen. Am Ende des zweiten Kapitels wird der Spracherkenner „*Janus Recognition Toolkit*" (JRTK, im Folgenden auch JANUS genannt) kurz vorgestellt und beschrieben, in welcher Konfiguration er in dieser Arbeit eingesetzt worden ist.
In Kapitel 3 und 4 werden die Optimierungen der Vorverarbeitung und der Decodierung in der Theorie beschrieben. Viele der hier vorgestellten Optimierungen beruhen auf der Vermeidung von Fließkommazahlen, die auf einem heutigen Handheld PC nur emuliert und somit unter großen Geschwindigkeitseinbußen zur Verfügung stehen. Diese Art der Optimierung lohnt sich immer dann, wenn viele Rechenoperationen auf wiederkehrenden Daten ausgeführt werden müssen, wie etwa bei der Matrixmultiplikation, der Fourier-Transformation (FFT) oder der Bewertungsfunktion während des Decodierungsvorgangs. Zur Optimierung der FFT wird auf die *Intel Performance Primitives* Funktions-Bibliothek zurückgegriffen.
Die Verwendung von Nachschlagetabellen (*engl. Look Up Table*) hat sich in dieser Arbeit als wenig effizient erwiesen. Das Problem hierbei ist der langsame Speicher und der kleine Daten-Cache eines PDA. Der zusätzliche Speicherzugriff bremst den Prozessor aus, der in der gleichen Zeit mehrere Ganzzahloperationen durchführen könnte.
Es werden aber auch Algorithmen vorgestellt, die eine Beschleunigung methodisch erzielen und auf anderen Systemen die gleiche Effizienz aufweisen. So kann durch die frühe Reduzierung der Merkmalsvektoren (*engl.* ***E**arly **F**eature **V**ector **R**eduction*, EFVR) die Geschwindigkeit deutlich gesteigert werden, da während der Decodierung weniger Merkmalsvektoren ausgewertet werden.
Zu jeder Optimierung werden einige Testergebnisse präsentiert, die einen Eindruck vermitteln, welche Beschleunigung und welcher Verlust der Wortakkuratheit jeweils zu erwarten sind. Dabei wurde weniger Wert auf eine hohe absolute Wortakkuratheit gelegt, als auf die praktische Durchführbarkeit der Evaluierung. So wurde auf Sprecher- und Kanaladaption während der Laufzeit komplett verzichtet, da aus Mangel an Speicherplatz des PDAs die Evaluierung nicht in einem Durchlauf durchgeführt werden konnte. Wichtiger ist es zu zeigen, dass die Wortakkuratheit erhalten bleibt oder nur wenig beeinträchtigt wird. Eine Adaption während der Laufzeit ist aber zusammen mit den Optimierungen grundsätzlich möglich. Das dazu verwendete akustische Modell wurde auf spontane Sprache in Telefonqualität (8kHz/16 Bit) trainiert. Die Testdaten wurden allerdings mit einer Abtastrate von 16kHz

aufgenommen und für diese Tests nachträglich konvertiert. Für die abschließenden Evaluierungen in Kapitel 6 wurde der grundsätzliche Aufbau des Erkenners beibehalten und die Testdaten mit 16kHz Abtastrate verwendet. Eine genauere Beschreibung der Testdaten wird dort gegeben.
Kapitel 5 beschäftigt sich mit der konkreten Implementierung der Optimierungen für den Spacherkenner JANUS. Zuerst musste der Spracherkenner auf das Betriebssystem Windows CE 3.0 portiert werden, um anschließend die Optimierungen einfügen und testen zu können. In Anhang A wird eine Dokumentation über alle Implementierungen gegeben.
In Kapitel 6 sind die abschließenden Evaluierungen mit allen Optimierungen aufgeführt, die sich als effizient erwiesen haben. Die Testdaten bestanden aus 363 Aufnahmen von englischsprachigen Äußerungen verschiedener Sprecher (4 Frauen und 5 Männer), die touristische Fragen über den Raum Karlsruhe stellen. Die zu erkennenden Texte stellen also eine Mixtur aus englischen Wörtern mit deutschen Straßennamen, Sehenswürdigkeiten und Namen von Hotels dar. Hierfür wurden zwei weitere, für diese Aufgabe verbesserte akustische Modelle verwendet. Da unter Verwendung der ursprünglichen Konfiguration des Erkenners auch mit den Optimierungen keine Echtzeit erreicht werden konnte, wurde mit Hilfe eines engeren Suchstrahls während der Decodierung die Geschwindigkeit auf Echtzeit getrimmt. Abschließend wird eine Zusammenfassung gegeben, um die Optimierungen zu bewerten und auf weiterführende Ansätze und Ideen zu verweisen.

2 Grundlagen

2.1 Limitierungen eines PDA

2.1.1 Limitierungen der Rechengeschwindigkeit

Ein PDA besitzt einen langsameren Prozessor als ein Desktop PC auf einem vergleichbaren Stand der Technik. Die Gründe hierfür sind hauptsächlich Energieersparnis, Wärmeentwicklung und die Kosten. Trotzdem erreichen die heutigen PDAs Geschwindigkeiten, wie sie vor einigen Jahren im Desktop Bereich zu finden waren. So befindet sich bei Erstellung dieser Arbeit der Stand der Technik bei ungefähr 400MHz Prozessortaktung, ausgestattet mit Techniken wie Befehls-Pipelining, Daten- und Befehls-Cache und Speicherzugriff im BurstMode[2], wie sie auch bei Prozessoren im Desktop Bereich zu finden sind. Es gibt aber einige zusätzliche Einschränkungen, die gerade bei der Spracherkennung Schwierigkeiten bereiten. So besitzt der Prozessor in der Regel keine Fließkommaeinheit (*engl.* ***F****loating* ***P****oint* ***U****nit*, FPU), die für das genaue Berechnen von Kommazahlen zuständig ist. Solche Fließkommazahlen können zwar für den Programmierer transparent verwendet werden, müssen aber durch eine Vielzahl von Rechenoperationen auf Ganzzahlen ersetzt werden. Tabelle 1 vermittelt hier einen Eindruck. Eine der wichtigsten und auf dem PDA effizientesten Methoden zur Beschleunigung ist daher das Ersetzen der Fließkommzahlen durch Ganzzahlen. Diese Methoden verlieren ihre Effizienz auf Systemen mit FPU. Dort muss im Einzelfall getestet werden, ob der zusätzliche Aufwand wie das Umwandeln der Fließkommazahlen in Ganzzahlen die Beschleunigung nicht überwiegt.

2.1.2 Limitierungen des Speichers

Der Arbeitsspeicher eines PDAs ist in der Regel um den Faktor vier bis 16 kleiner als der eines vergleichbaren Desktop PCs. Eine weitere große Einschränkung ergibt sich aus dem Fehlen eines Massenspeichers, den moderne Systeme zum Erweitern des Arbeitsspeichers nutzen. Daraus ergibt sich auch der große Nachteil, dass alle Daten im Arbeitsspeicher auf einem virtuellen RAM-Laufwerk[3] gehalten werden müssen, sofern keine Erweiterungskarte zur Verfügung steht. Etwas abgemildert wird diese Einschränkung dadurch, dass die meisten Betriebssysteme solche Daten transparent für den Benutzer mit einem verlustfreien Verfahren packen.

[2] Im sog. BurstMode können durch eine Optimierung des Speicherchips mehrere Datenwörter in Reihe schneller gelesen werden.

[3] Ein RAM-Laufwerk verhält sich für den Benutzer wie eine normale Festplatte, die Daten werden aber im Arbeitsspeicher (***R****andom* ***A****ccess* ***M****emory*, RAM) abgelegt.

Code ohne FPU:	Code mit FPU:

MOVEM.L D3-D5,-(A7)
MOVE.B D0,D5
BEQ.S label1
MOVE.B D1,D4
BEQ.S label2
ADD D5,D5
ADD D4,D4
MOVEQ #-$80,D3
EOR.B D3,D4
EOR.B D3,D5
ADD.B D4,D5
BVS.S label6
MOVE.B D3,D4
EOR D4,D5
ROR #1,D5
SWAP D5
MOVE D1,D5
CLR.B D0
CLR.B D5
MOVE D5,D4
MULU D0,D4
SWAP D4
MOVE.L D0,D3
SWAP D3
MULU D5,D3
ADD.L D3,D4
SWAP D1
MOVE.L D1,D3
MULU D0,D3
ADD.L D3,D4
CLR D4
ADDX.B D4,D4
SWAP D4
SWAP D0
MULU D1,D0
SWAP D1
SWAP D5
ADD.L D4,D0
BPL.S label4
ADD.L #$80,D0
MOVE.B D5,D0
BEQ.S label2
label1:
MOVEM.L (A7)+,D3-D5
RTS
label4:
SUBQ.B #1,D5
BVS.S label2
BCS.S label2
MOVEQ #$40,D4
ADD.L D4,D0
ADD.L D0,D0
BCC.S label5
ROXR.L #1,D0
ADDQ.B #1,D5
label5:
MOVE.B D5,D0
BEQ.S label2
MOVEM.L (A7)+,D3-D5
RTS
label2:
MOVEQ #0,D0
MOVEM.L (A7)+,D3-D5
RTS
label6:
BPL.S label2
EOR.B D1,D0
OR.L #$FFFFFF7F,D0
TST.B D0
ORI #2,CCR
MOVEM.L (A7)+,D3-D5
RTS
	FMUL.f fp0,fp1

Tabelle 1: 68K-Assembler-Code einer Fließkomma-Multiplikation ohne und mit FPU

2.2 Verwandte Arbeiten

Diese Arbeit stützt sich hauptsächlich auf Konferenzbände der *International Conference on Spoken Language Processing* (ICSLP), *Automatic Speech Recognition and Understanding Workshop* (ASRU) und *International Conference on Acoustics, Speech, and Signal Processing* (ICASSP) der letzen vier Jahre. Das Thema Optimierung für den Handheld PC im Zusammenhang mit automatischer Spracherkennung ist relativ neu, sodass bisher wenig Standard-Literatur zu finden ist. Auch eine Recherche im Internet bringt kaum Erkenntnisse, da dieses Thema Gegenstand der industriellen Forschung ist und nur wenige Details veröffentlicht werden.
Viele Abhandlungen in diesem Zusammenhang beschäftigen sich mit der Reduzierung des Speicheraufwands, der neben der Geschwindigkeit die größte Limitierung darstellt [Tia04]. Oft kann aber indirekt mit einem kompakteren akustischen Modell, einer Grammatik oder einem Vokabular eine Beschleunigung des Erkennungsvorgangs erzielt werden. In [PK04] werden die *State-Clustered Tied-Mixture* (SCTM) HMMs vorgestellt, die eine Verkleinerung des akustischen Modells mit gleichzeitiger Beschleunigung der Decodierung bewirken. Nach [Nov04] ist ein wichtiger, limitierender Faktor für die Erkennung der langsame Speicher eines Handheld PCs, was auch in dieser Arbeit nachvollzogen werden konnte. Auch [JKO04] beschäftigt sich mit der kompakteren Darstellung des akustischen Modells, um bei gleichbleibendem Speicheraufwand die Wortakkuratheit zu erhöhen. Speicheroptimierungen werden aber in dieser Arbeit nicht betrachtet, auch wenn sie ein wichtiger Teil der Optimierung für Handheld PCs darstellen.
Im Zusammenhang mit eingebetteten oder mobilen Systemen beschäftigen sich viele Abhandlungen mit Optimierungen, die auf wesentlich kleinere Systeme zugeschnitten sind als der Handheld PC, oder kleinere Erkenner ohne Grammatiken behandeln wie das Wählen von Telefonnummern oder Empfangen von Kommandos auf einem Mobiltelefon [JHJK04]. In dieser Arbeit wird aber versucht, den Erkenner ohne Einschränkung der Kenngrößen und unter Beibehaltung der freien Konfigurierbarkeit zu optimieren, sodass er leicht als Ausgangsbasis für weitere Studien dienen kann.
Eine gute Einführung über die Problemstellungen der Spracherkennung auf mobilen Geräten wird in [Vii01] gegeben. Hier werden die verschiedenen grundsätzlichen Lösungsansätze beleuchtet, wie zum Beispiel die Erkennung über eine Kommunikationsverbindung zu einem Server, der die Spracherkennung durchführt und das Ergebnis als Text zurück sendet.
Mit der eigentlichen Beschleunigung des Erkennungsvorgangs auf dem PDA beschäftigt sich [VISV04], aus dem die Optimierung der Decodierung mit Hilfe von Nachschlagetabellen entnommen ist. In [ZGS+03] wird die Verwendung von Ganzzahlarithmetik und die Bestimmung von Skalierungsfaktoren erwähnt. Auch speziell die Verwendung der *Intel Performance Primitives* Funktions-Bibliothek wird vorgeschlagen. Eine Beschreibung der Testdaten,

die für die Evaluierung verwendet wurden, kann in [FSS+03] gefunden werden. Die beiden in der abschließenden Evaluierung verwendeten akustischen Modelle basieren auf den Trainingsdaten, die in [MJF+04] beschrieben werden.
Zur Kategorisierung der Optimierungen der Decodierung wird das Vier-Schichten-Modell aus [CSMR04] aufgegriffen. Es wird um eine fünfte Schicht erweitert, um den Optimierungen auf Ausführungsebene in dieser Arbeit besser zu entsprechen. Eine Beschreibung des JANUS Spracherkenners und der bereits vor dieser Arbeit implementierten Optimierungen wie dem *Bucket Box Intersection Tree* (BBI Tree) sind in [Wos98] zu finden. Hier wurde auch der Ansatz für die EFVR entnommen, die auf dem *Conditional Frame Skipping* aufbaut. Erfahrungen über die schlechte Aufnahmequalität eines PDAs und eine Anwendung von Sprache-zu-Sprache-Übersetzung ist in [WBB+03] festgehalten. Die Spezifikationen der Soft- und Hardware für die Spracherkennung ist sehr ähnlich zu dieser Arbeit.

2.3 Überblick über den Spracherkenner JANUS

In dieser Arbeit wurde als Spracherkenner das „*Janus Recognition Toolkit*" (JRTK, [FGH+97]) mit dem „*Ibis*" Decoder [SMFW01] der Universität Karlsruhe verwendet. JANUS besitzt keinen festen Verarbeitungsablauf, sondern kann mit Hilfe der Skriptsprache TCL[4] frei programmiert werden [Wel97]. Diese Art der Implementierung eines Spracherkenners eignet sich besonders gut für Studienzwecke, da sehr einfach neue Algorithmen oder verschiedene Abläufe realisiert werden können.
Der Signalverlauf des konkreten Erkenners, wie er in dieser Arbeit verwendet wurde, ist schematisch in Abbildung 1 dargestellt. Er basiert auf der MFCC (***M**el-**F**requency **C**epstral **C**oefficient*) Vorverarbeitung, die in [YW00] vorgestellt wird. Das Audio-Signal wird digitalisiert und steht als Eingabe zur Verfügung. Es wird in Zeitfenster eingeteilt und in zwei Wege aufgeteilt. Der erste Weg bereitet das Signal für die Sprach-Detektion auf, die zu erkennen versucht, ob ein Zeitfenster Sprachsignale enthalten könnte oder nicht (Normalize, Filter, ALog). Damit werden später die Merkmalsvektoren angepasst (MeanSub).
Wichtiger ist aber der zweite Weg. Jedes Zeitfenster bildet einen Merkmalsvektor. Das Signal in jedem Zeitfenster wird mit Hilfe der Fast-Fourier-Transformation (FFT) in das Leistungsspektrum überführt. Danach findet eine Vokal-Trakt-Längen-Normalisierung (VTLN) statt, die allerdings nicht sprecherabhängig durchgeführt wird, sondern mit einem festen Warp-Faktor arbeitet, der über den gesamten Trainingsdaten ermittelt wurde. Dadurch trägt die VTLN wenig zu einer besseren Erkennung bei, aber sie wird bei der Betrachtung der Vorverarbeitung mit einbezogen. Mit Hilfe der Filter-

[4] TCL steht für „***T**ool **C**ommand **L**anguage*" und ist eine OpenSource Skriptsprache.

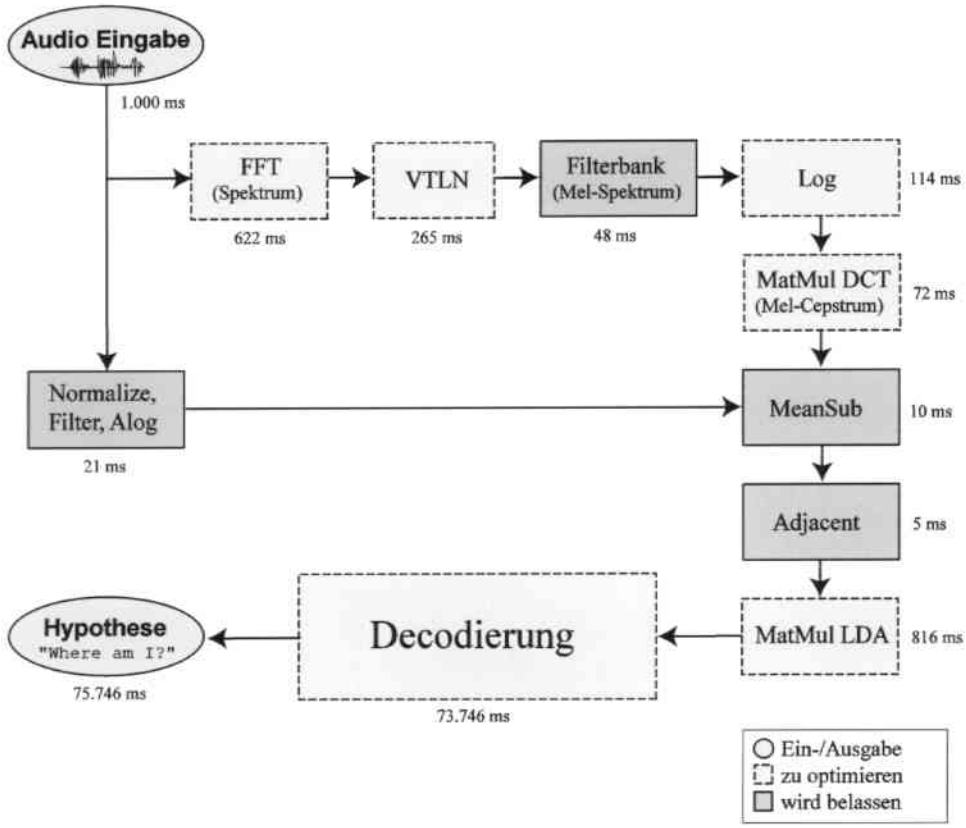

Abbildung 1: Verweildauer in jedem Verarbeitungsschritt für 1s Audio Signal

bank wird das Spektrum in eine Mel-Skala überführt, die den Frequenzverteilungen der menschlichen Sprache besser entspricht als die lineare Skala der Fourier-Transformation. Danach werden die Koeffizienten durch Logarithmierung (Log) mit anschließender DCT[5] in Form einer Matrixmultiplikation (MatMul DCT) in das Mel-Cepstrum transformiert. Jetzt werden die beiden Signalwege wieder zusammengeführt, und die Merkmalsvektoren werden mit Hilfe der Sprach-Detektion angepasst (MeanSub). Hier findet in anderen Konfigurationen auch die Sprecher- und Kanalanpassung statt. Um den Kontext eines Merkmalsvektors mit einzubeziehen, wird jeder Vektor mit den Koeffizienten seiner Nachbarvektoren erweitert (Adjacent). Die letzte Matrixmultiplikation (MatMul LDA) führt eine lineare Diskriminanzanalyse durch, um die Distanzen zwischen den Merkmalsklassen zu erhöhen, und die Distanzen innerhalb einer Merkmalsklasse zu verringern. Dabei wird gleichzeitig der Merkmalsraum auf 32 Dimensionen reduziert. Damit ist die Vorverarbeitung abgeschlossen und die Merkmalsvektoren werden in die Decodierung gegeben, die anschließend die beste gefundene Hypothese für den gesprochenen Text ausgibt.
In Abbildung 1 ist auch die Verweildauer in jedem Verarbeitungsschritt di-

[5]**D**iskrete **C**osinus **T**ransformation, DCT

rekt nach der Portierung auf den PDA eingetragen. Die markierten Verarbeitungsschritte werden im Verlauf dieser Arbeit betrachtet und jeweils eine oder mehrere Optimierungen vorgestellt. Die Zeitangaben repräsentieren jeweils die durchschnittlich benötigte Bearbeitungszeit für 1s Audio-Eingabe auf den gesamten Testdaten. Später kann daraus sehr leicht der Echtzeitfaktor bestimmt werden. Als Ausgangsbasis (*engl. Baseline*) während der Testphase der einzelnen Optimierungen benötigte der Erkenner auf dem PDA 75,7s für eine Sekunde Audio-Eingabe. Daraus ergibt sich direkt der Echtzeitfaktor von 75,7. Die Wortakkuratheit lag bei 69,27%. Die Testergebnisse in Kapitel 3 und 4 werden mit dieser Ausgangsbasis verglichen. In der abschließenden Evaluierung wurde eine leicht veränderte Konfiguration und zwei weitere akustische Modelle verwendet, die in der Ausgangsbasis 52,3s bzw. 52,7s benötigten. Die Wortakkuratheit lag bei 73,36% bzw. 73,77%.
Um die Zeitintervalle messen zu können, wurde eine Meßfunktion implementiert, die in einer zeitlichen Auflösung von 1ms arbeitet. Da die Zeit für jede der 363 Äußerung einzeln gemessen wurde, kann sich maximal ein absoluter Fehler von 363ms addieren. Bei Funktionen, die für jeden Merkmalsvektor einer Äußerung aufgerufen werden, kann der Fehler sogar um den Faktor 100-300 höher liegen. Um diese Ungenauigkeit zu kompensieren, wurden die Ergebnisse von kurzen Zeitintervallen mehrmals gemessen und der Mittelwert gebildet.

3 Optimierung der Vorverarbeitung

Nach Abbildung 1 aus Kapitel 2 scheint der Aufwand für die Vorverarbeitung im Vergleich zur Decodierung zunächst sehr gering zu sein. Die Decodierung kann aber durch Variation der Strahlbreite, der Größe der Akustik, des Vokabulars und der Grammatik stark beeinflusst werden. Die Vorverarbeitung bietet diesen Spielraum nicht.
Die Ausgangsbasis, mit der die Optimierungen verglichen werden, benötigte für die Vorverarbeitung bereits einen Echtzeitfaktor von über 2,0. Eine Spracherkennung in Quasi-Echtzeit wäre damit von vornherein nicht möglich. Darum ist auch hier eine starke Beschleunigung wichtig.
Im Folgenden werden alle Schritte der Vorverarbeitung betrachtet, die länger als 100ms benötigten, und es wird jeweils ein Vergleich mit der Ausgangsbasis gegeben. Eine Ausnahme ist die VTLN, die ausschließlich durch Optimierung des Quellcodes beschleunigt wurde und im theoretischen Teil nicht weiter behandelt wird.

3.1 Matrixmultiplikation mit Ganzzahlarithmetik

Zu beschleunigen ist die Matrixmultiplikation

$$\mathbf{C} = \mathbf{A} * \mathbf{B}$$

mit den gegebenen Matrizen $\mathbf{A}^{(m_A \times n)}$ und $\mathbf{B}^{(n \times m_B)}$ und der Ergebnismatrix $\mathbf{C}^{(m_A \times m_B)}$. Die Matrixmultiplikation besitzt den Aufwand $O(n^3)$.

Es existieren einige, wohl bekannte Methoden zur Verbesserung des Aufwands, wie die Methode von Strassen oder die Methode von Coppersmith und Winograd. Diese Methoden zielen jedoch darauf ab, die Anzahl der Multiplikationen zu reduzieren und dabei eine Erhöhung der Anzahl der Additionen in Kauf zu nehmen, in der Annahme, Multiplikationen seien teurer als Additionen. Da die Software-Implementierung der Fließkomma-Addition aber nicht deutlich schneller ist als die Multiplikation, scheiden diese Methoden zur Beschleunigung der Matrixmultiplikation auf dem PDA aus.

Die naive Implementierung besteht neben den Schleifen Anweisungen fast ausschließlich aus Fließkomma-Operationen. Sie eignet sich daher besonders für eine Beschleunigung durch Verwendung von Ganzzahlarithmetik.
Die grundsätzliche Vorgehensweise ist dabei denkbar einfach: Man wandelt die beiden Matrizen aus Fließkommazahlen in zwei Matrizen aus Ganzzahlen um. Danach wird die Matrixmultiplikation mit der naiven Implementierung durchgeführt und das Ergebnis wieder in Fließkommazahlen zurück gewandelt. Dabei verliert man aber die Nachkommastellen komplett und erhält einen Abrundungsfehler, der in Abhängigkeit des Wertebereichs der Matrizen liegt. Im schlimmsten Fall bewegen sich alle Werte unterhalb von 1.0,

und das Resultat ist eine Null-Matrix.
Daher skaliert man die Koeffizienten von **A** und **B** jeweils mit einem Faktor s_A und s_B, wandelt sie erst dann in Ganzzahlen um und führt die Matrixmultiplikation durch. Nach der Rückwandlung in Fließkommazahlen multipliziert man die Ergebnismatrix mit dem Kehrwert des Produktes $s_A \cdot s_B$, um die Skalierung wieder rückgängig zu machen.

$$\mathbf{C} \approx \frac{1}{s_A s_B} (\lfloor s_A \cdot \mathbf{A} \rfloor * \lfloor s_B \cdot \mathbf{B} \rfloor) \quad (1)$$

Die Gaußklammern $\lfloor \ldots \rfloor$ symbolisieren hier die Umwandlung einer Fließkommazahl in eine Ganzzahl, bei der üblicherweise sowohl bei positiven als auch negativen Zahlen die Nachkommastelle einfach abgeschnitten wird.
Ziel ist es nun, möglichst große Skalierungsfaktoren $s_A > 0$ und $s_B > 0$ zu finden, um den Zahlenbereich der Ganzzahlen $[INT_{min}, INT_{max}]$ gut auszuschöpfen, ohne dabei aber einen Überlauf zu produzieren, der das Ergebnis komplett verfälschen würde.

3.1.1 Vorüberlegungen für die Bestimmung von s_A und s_B

Ein Koeffizient c_{ij} der Ergebnis-Matrix **C** berechnet sich wie folgt:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Skaliert mit $s_A > 0$ und $s_B > 0$ ergibt sich:

$$(s_A s_B) \cdot c_{ij} = \sum_{k=1}^{n} (s_A \cdot a_{ik})(s_B \cdot b_{kj})$$

wobei n die Anzahl der Spalten von **A** und die Anzahl der Zeilen von **B** ist. Um einen Ganzzahl-Überlauf zu verhindern, muss garantiert werden, dass $\forall i, j, k$:

$$\begin{aligned} (s_A s_B) \cdot c_{ij} &\in [INT_{min}, INT_{max}] \\ (s_A \cdot a_{ik})(s_B \cdot b_{kj}) &\in [INT_{min}, INT_{max}] \end{aligned}$$

Da die Ganzzahlen in Zweierkomplement-Darstellung vorliegen, dürfen Zwischenergebnisse der Summation überlaufen, sie beeinträchtigen das Ergebnis nicht.

Im Folgenden wird zur Vereinfachung davon ausgegangen, dass alle Koeffizienten von **A** und **B** positiv sind. Treten negative Koeffizienten auf, gilt offensichtlich

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \leq \sum_{k=1}^{n} |a_{ik}||b_{kj}|$$

und die Abschätzung von c_{ij} nach oben wäre somit auch korrekt.
Bei ausschließlich positiven Koeffizienten genügt es nun tatsächlich, nur c_{ij} nach oben abzuschätzen, weil c_{ij} immer größer ist als der größte Summand, aus dem es sich zusammensetzt:

$$c_{ij} \geq max_{1 \leq k \leq n}\{a_{ik}b_{kj}\}$$

Man fordert also:

$$\max\{(s_A s_B) \cdot c_{ij}\} \leq INT_{max} \tag{2}$$

Da im Zweierkomplement-Raum $|INT_{max}| \leq |INT_{min}|$ gilt, haben wir den Zahlenbereich gleichzeitig auch nach unten abgeschätzt.

Für alle Skalierungsfaktoren s gelte im Folgenden $s > 0$.

3.1.2 Abschätzung eines gemeinsamen Skalierungsfaktors s

Zunächst wird zur Vereinfachung für **A** und **B** ein gemeinsamer Skalierungsfaktor $s = s_A = s_B$ gesucht, der die Bedingung (2) erfüllt.

$$\begin{aligned}
\max\{s^2 \cdot c_{ij}\} &\leq INT_{max} \\
\max_{1 \leq i \leq m_A, 1 \leq j \leq m_B} \left\{ \sum_{k=1}^{n} (s \cdot a_{ik})(s \cdot b_{kj}) \right\} &\leq INT_{max} \\
s^2 \cdot \max_{1 \leq i \leq m_A, 1 \leq j \leq m_B} \left\{ \sum_{k=1}^{n} a_{ik} b_{kj} \right\} &\leq INT_{max} \\
\Leftarrow s^2 \cdot \sum_{k=1}^{n} \max_{1 \leq i \leq m_A} \{a_{ik}\} \cdot \max_{1 \leq j \leq m_B} \{b_{kj}\} &\leq INT_{max} \\
\Leftarrow s^2 \cdot n \cdot \max\{a_{ik}\} \cdot \max\{b_{kj}\} &\leq INT_{max} \\
s^2 &\leq \frac{INT_{max}}{n \cdot \max\{a_{ik}\} \cdot \max\{b_{kj}\}} \\
s &\leq \sqrt{\frac{INT_{max}}{n \cdot \max\{a_{ik}\} \cdot \max\{b_{kj}\}}}
\end{aligned}$$

Die Bestimmung von $\max\{a_{ik}\}$ und $\max\{b_{kj}\}$ benötigt $O(n^2)$ Vergleiche, kann jedoch in vertretbarer Zeit durchgeführt werden, da Vergleichsoperationen auf Fließkommazahlen auch ohne FPU wesentlich schneller sind als Rechenoperationen.
Somit ist ein Skalierungsfaktor s gefunden, mit dem die Matrixmultiplikation in (1) berechnet werden kann, ohne einen Überlauf im Ganzzahlraum zu riskieren. Alle im Folgenden vorgestellten Methoden unterscheiden sich lediglich in einer verfeinerten Abschätzung der Skalierungsfaktoren.

3.1.3 Abschätzung von zwei Skalierungsfaktoren s_A und s_B

Verwendet man den gleichen Skalierungsfaktor s für beide Matrizen, wird nicht der Tatsache Rechnung getragen, dass die Matrizen **A** und **B** sehr unterschiedliche Wertebereiche haben können.

Ein Beispiel:
Befindet sich ein Koeffizient der Matrix **A** über INT_{max}, so erhält man einen Skalierungsfaktor von $s < 1$. Liegt der Wertebereich der Koeffizienten von Matrix **B** zwischen 0 und 1, so werden diese Koeffizienten nicht höher skaliert und im ganzzahligen Raum zu 0 abgerundet. Die Ergebnismatrix **C** wird zur Null-Matrix.

Skaliert man die beiden Matrizen **A** und **B** mit verschiedenen Faktoren s_A und s_B, sodass die Maxima beider Matrizen auf den gleichen Wert skaliert werden, kann die Matrixmultiplikation aus dem obigen Beispiel sinnvoll durchgeführt werden. Die Genauigkeit ist dann sogar unabhängig von den Wertebereichen der Matrizen und hängt nur von der Verteilung der Koeffizienten im Wertebereich ab. Diese Methode ist deshalb immer vorzuziehen, wenn die beiden Matrizen keinen sehr ähnlichen Wertebereich besitzen.
Nach Bedingung (2) muss wieder gelten:

$$\begin{aligned}
\max\{(s_A s_B) \cdot c_{ij}\} &\leq INT_{max} \\
\max_{1\leq i\leq m_A, 1\leq j\leq m_B} \left\{ \sum_{k=1}^{n} (s_A \cdot a_{ik})(s_B \cdot b_{kj}) \right\} &\leq INT_{max} \\
\Leftarrow \sum_{k=1}^{n} \max_{1\leq i\leq m_A} \{s_A \cdot a_{ik}\} \max_{1\leq j\leq m_B} \{s_B \cdot b_{kj}\} &\leq INT_{max} \\
\Leftarrow n \cdot \max\{s_A \cdot a_{ik}\} \cdot \max\{s_B \cdot b_{ki}\} &\leq INT_{max}
\end{aligned}$$

Wir fordern nun:

$$\max\{s_A \cdot a_{ik}\} =^{(!)} \max\{s_B \cdot b_{ki}\}$$

um für beide Matrizen den gleichen Zahlenraum zur Verfügung zu stellen.
Somit ergibt sich für s_A:

$$\begin{aligned}
n \cdot \max\{s_A \cdot a_{ik}\} \cdot \max\{s_A \cdot a_{ik}\} &\leq INT_{max} \\
{s_A}^2 &\leq \frac{INT_{max}}{n \cdot \max\{a_{ik}\}^2} \\
s_A &\leq \sqrt{\frac{INT_{max}}{n \cdot \max\{a_{ik}\}^2}}
\end{aligned}$$

Und analog ergibt sich s_B:

$$s_B \leq \sqrt{\frac{INT_{max}}{n \cdot \max\{b_{kj}\}^2}}$$

3.1.4 Verbesserte Abschätzung eines gemeinsamen Skalierungsfaktors s

Eine echte Verbesserung der Abschätzung von s erhält man, wenn die Maxima aller Zeilen der Matrix **A** und die Maxima aller Spalten der Matrix **B** berechnet werden und somit der letzte abschwächende Schritt der obigen Abschätzung nicht vollzogen werden muss:

$$\begin{aligned}
\max\{s^2 \cdot c_{ij}\} &\leq INT_{max} \\
\max_{1\leq i\leq m_A, 1\leq j\leq m_B} \left\{ \sum_{k=1}^{n} (s \cdot a_{ik})(s \cdot b_{kj}) \right\} &\leq INT_{max} \\
s^2 \cdot \max_{1\leq i\leq m_A, 1\leq j\leq m_B} \left\{ \sum_{k=1}^{n} a_{ik} b_{kj} \right\} &\leq INT_{max} \\
\Leftarrow s^2 \cdot \sum_{k=1}^{n} \max_{1\leq i\leq m_A} \{a_{ik}\} \max_{1\leq j\leq m_B} \{b_{kj}\} &\leq INT_{max}
\end{aligned}$$

$$\begin{aligned}
s^2 &\leq \frac{INT_{max}}{\sum_{k=1}^{n} \max_{1\leq i\leq m_A}\{a_{ik}\} \max_{1\leq j\leq m_B}\{b_{kj}\}} \\
s &\leq \sqrt{\frac{INT_{max}}{\sum_{k=1}^{n} \max_{1\leq i\leq m_A}\{a_{ik}\} \max_{1\leq j\leq m_B}\{b_{kj}\}}}
\end{aligned}$$

Die Berechnung der Zeilen- und Spaltenmaxima benötigt keinen zusätzlichen Speicheraufwand, da die Summe noch während der Suche nach den Maxima berechnet werden kann, und die Maxima später nicht mehr benötigt werden. Allerdings werden neben den $O(n^2)$ Vergleichen auch noch $O(n)$ Fließkomma-Additionen und Multiplikationen benötigt, weshalb die Berechnung von s_A und s_B etwas langsamer ist als bei der einfachen Abschätzung mit matrix-globalen Maxima. Diese Abschätzung erhöht jedoch deutlich den Skalierungsfaktor bei Matrizen mit heterogenen Koeffizienten und somit die Genauigkeit der Ergebnismatrix **C**.

3.1.5 Verbesserte Abschätzung von zwei Skalierungsfaktoren s_A und s_B

Die Aufteilung von s in zwei verschiedene Skalierungsfaktoren s_A und s_B kann auch bei der verbesserten Abschätzung angewandt werden. Allerdings

kann man hier nicht mehr die Maxima der Matrizen gleichsetzen, da mit der Summe der Spalten- und Zeilenmaxima abgeschätzt wird. Daher berechnet man zuerst s mit der verbesserten Abschätzung, und teilt s anschließend auf:

$$s_A s_B = s^2$$

Das Verhältnis der Skalierungsfaktoren wird von der einfachen Abschätzung übernommen, dort galt:

$$\begin{aligned} s_A &\leq \sqrt{\frac{INT_{max}}{n \cdot \max\{a_{ik}\}^2}} \\ &= \sqrt{\frac{INT_{max}}{n \cdot \max\{a_{ik}\}^2} \cdot \frac{max\{b_{kj}\}}{max\{b_{kj}\}}} \\ &= \sqrt{\frac{INT_{max}}{n \cdot \max\{a_{ik}\} \max\{b_{jk}\}} \cdot \frac{max\{b_{kj}\}}{max\{a_{ik}\}}} \\ &= s \cdot \sqrt{\frac{max\{b_{kj}\}}{max\{a_{ik}\}}} \end{aligned}$$

Analog zu der einfachen Abschätzung erhält man:

$$s_A = s \cdot \sqrt{\frac{max\{b_{kj}\}}{max\{a_{ik}\}}}$$

und

$$s_B = s \cdot \sqrt{\frac{max\{a_{ik}\}}{max\{b_{kj}\}}}$$

3.1.6 Skalierung mit Faktoren für jede Zeile und Spalte

Für die einfache Abschätzung mit zwei verschiedenen Skalierungsfaktoren s_A und s_B ist nur die jeweils dazugehörige Matrix notwendig. Daher kann noch einen Schritt weiter gegangen werden, und für jede Zeile der Matrix **A** und jede Spalte der Matrix **B** ein eigener Skalierungsfaktor s_{Ai} und s_{Bj} bestimmt werden, als wäre es jeweils eine einzeilige bzw. einspaltige Matrix. Ausser in der Zeile bzw. Spalte, in der sich der Koeffizient mit dem höchsten Wert befindet, ist der so gefundene Skalierungsfaktor höher als bei der einfachen Abschätzung von oben. Die Matrixmultiplikation wird dadurch mindestens gleich genau oder genauer durchgeführt. Analog zu 3.1.3 ergibt sich als Abschätzung für eine einzelne Zeile i von **A** der Faktor s_{Ai}:

$$s_{Ai} \leq \sqrt{\frac{INT_{max}}{n \cdot \max_{1 \leq k \leq n}\{a_{ik}\}^2}}$$

Und analog für s_{Bj}:

$$s_{Bj} \leq \sqrt{\frac{INT_{max}}{n \cdot \max_{1 \leq k \leq n} \{b_{kj}\}^2}}$$

Die Berechnung der Matrixmultiplikation sei veranschaulicht durch:

$$\begin{array}{c} \\ s_{A1} \rightarrow \\ s_{A2} \rightarrow \\ \vdots \\ s_{An} \rightarrow \end{array} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \hline \vdots & & & \\ a_{m_A 1} & a_{m_A 2} & \cdots & a_{m_A n} \end{pmatrix} * \begin{array}{c} \begin{array}{cccc} s_{B1} & s_{B2} & \cdots & s_{Bm} \end{array} \\ \begin{array}{cccc} \downarrow & \downarrow & & \downarrow \end{array} \\ \left(\begin{array}{c|c|c|c} b_{11} & b_{12} & \cdots & b_{1m_B} \\ b_{21} & b_{22} & \cdots & b_{2m_B} \\ \vdots & & \vdots & \\ b_{n1} & b_{n2} & \cdots & b_{nm_B} \end{array}\right) \end{array}$$

$$= \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1m_B} \\ c_{21} & \boxed{c_{22}} & \cdots & c_{2m_B} \\ \vdots & & & \\ c_{m_A 1} & c_{m_A 2} & \cdots & c_{m_A m_B} \end{pmatrix}$$

wobei sich die c_{ij} wie folgt berechnen:

$$(s_{Ai} s_{Bj}) \cdot c_{ij} = \sum_{k=1}^{n} (s_{Ai} \cdot a_{ik})(s_{Bj} \cdot b_{kj})$$

Diese Methode der Matrixmultiplikation hat allerdings einen zusätzlichen Speicheraufwand von $O(m_A + m_B)$, da die Skalierungsfaktoren zwischengespeichert werden müssen.

3.1.7 Evaluierung

In Tabelle 2 sind alle Versionen der Matrixmultplikationen und deren Einfluß auf die Wortakkuratheit aufgeführt. Daraus kann abgelesen werden, dass jede Abschätzung genau genug ist, um die Wortakkuratheit nicht zu beeinträchtigen. Zur Evaluierung wurde die LDA (MatMul LDA) mit Ganzzahlarithmetik durchgeführt. Der durchschnittliche und maximale Fehler eines Koeffizienten ergibt sich durch den Vergleich mit der Matrix, die mit Fließkommazahlen berechnet wurde. Die genaue Berechnung des durchschnittlichen und maximalen Fehlers ist im Anhang A.8 zu finden.
Es wurde auch die DCT des Spektrums in das Cepstrum (MatMul DCT) mit Ganzzahlarithmetik durchgeführt. Hier entsteht ebenfalls keine Beeinträchtigung der Wortakkuratheit.
In Abbildung 2 und 3 werden die verschiedenen Abschätzungen noch einmal grafisch gegenüber gestellt. Alle Versionen mit Ganzzahlarithmetik besitzen die gleiche Geschwindigkeit für die Durchführung der Matrixmultiplikation.

Sie unterscheiden sich allerdings in dem zusätzlichen Aufwand (*engl. Overhead*) für die Bestimmung der Skalierungsfaktoren. Ist der Wertebereich vorhersagbar, so können die Skalierungsfaktoren geschätzt und fest angegeben werden, wie es bei der Version „int_fix" der Fall ist. Hier wurden die fixen Skalierungsfaktoren aufgrund der Beobachtungen der Version „int_es" bestimmt, da diese Abschätzung die beste Genauigkeit liefert. Die Version „int_fix" wurde später in der abschließenden Evaluierung eingesetzt, da sie ausreichend genau ist und keinen zusätzlichen Aufwand für die Bestimmung der Skalierungsfaktoren benötigt. Während der Evaluierung war der Spielraum groß genug, dass keine Überläufe aufgetreten sind.
Die Methode „int_ra" mit einzelnen Skalierungsfaktoren für jede Zeile und Spalte überzeugt in diesem Fall nicht, da die Koeffizienten in den Matrizen während der Vorverarbeitung recht gleichmäßig verteilt sind. Bei Matrizen mit sehr unterschiedlichen Koeffizienten verbessert sich hier allerdings die Genauigkeit.
Abkürzungen für die verschiedenen Matrixmultiplikationen:

float : Matrixmultiplikation mit Fließkommazahlen
int_n : einfache Abschätzung mit einem Skalierungsfaktor s
int_ns : einfache Abschätzung mit zwei Skalierungsfaktoren s_A und s_B
int_e : verbesserte Abschätzung mit einem Skalierungsfaktor s
int_es : verbesserte Abschätzung mit zwei Skalierungsfaktoren s_A und s_B
int_ra : einfache Abschätzung mit einem Skalierungsfaktor für jede Spalte und Zeile
int_fix : fest bestimmte Skalierungsfaktoren s_A und s_B

Funktion	float	int_n	int_ns	int_e	int_es	int_ra	int_fix
Zeit [ms]	816	71	71	82	82	79	67
d. Fehler [%]	0	0,11	0,10	0,06	0,05	0,07	0,07
max. Fehler [%]	0	0,69	0,57	0,36	0,31	0,44	0,39
WA [%]	69,27	69,27	69,27	69,27	69,27	69,27	69,27

Tabelle 2: Benötigte Zeit, Fehler und Einfluß auf die Wortakkuratheit der Matrixmultiplikation

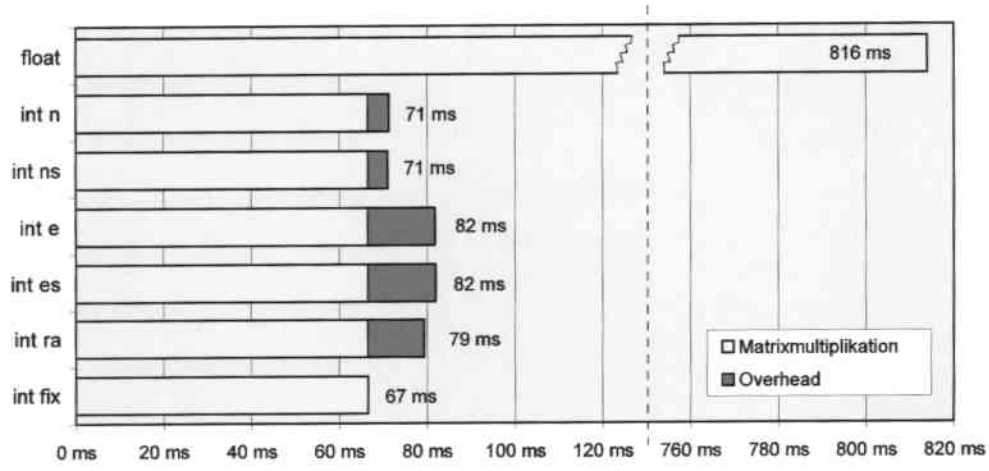

Abbildung 2: Benötigte Zeit der Matrixmultiplikationen im Vergleich

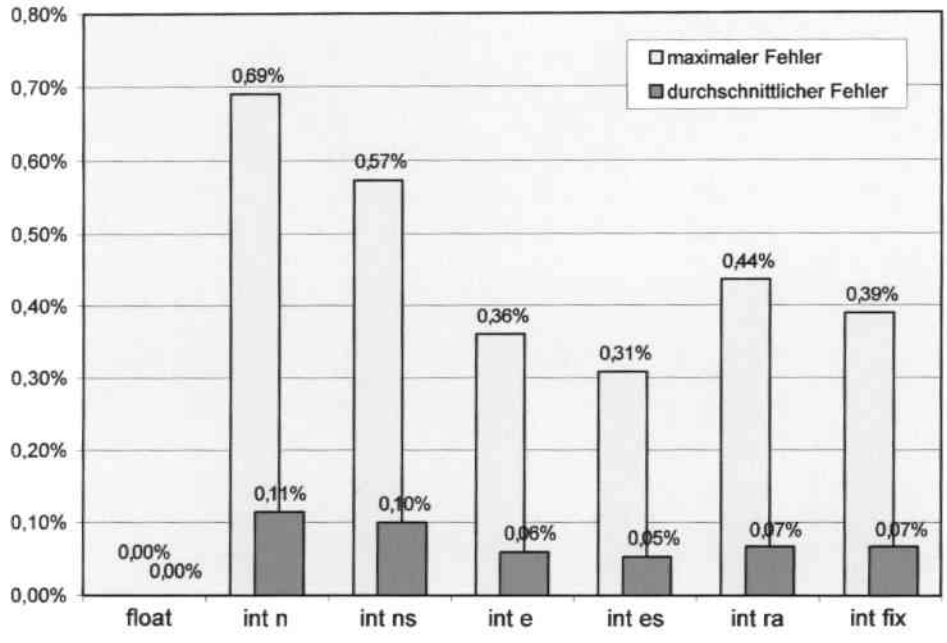

Abbildung 3: Genauigkeit der Matrixmultiplikationen im Vergleich

3.2 FFT mit Intel Performance Primitives

Die Fast-Fourier-Transformation (FFT) zur Gewinnung des Spektrums eines Sprachsignals ist eine wichtige Schlüsselfunktion zur Berechnung der Merkmalsvektoren. Sie besitzt den wohl bekannten Aufwand $O(n \log n)$, wobei der konstante Aufwand pro Ausführungsschritt sehr hoch ist. Auf dem PDA kommt hier wiederum besonders zum Tragen, dass für die Berechnung der FFT in der Regel Fließkommazahlen verwendet werden, die ohne eine FPU sehr langsam sind. Es wäre also naheliegend, auch die FFT mittels Ganzzahlarithmetik durchzuführen.
Für die Beschleunigung von Standardfunktionen wie die FFT gibt es aber bereits eine Reihe von Implementierungen, auf die man zurückgreifen kann. So gibt es von der Firma Intel eine auf Intel Prozessoren optimierte Befehlsbibliothek, die eine Reihe nützlicher Grundfunktionen für die Signalverarbeitung enthält, die *Intel Performance Primitives* (IPP) [ZGS+03]. Zu den unterstützten Prozessoren gehört neben der x86 Familie auch die Reihe der ARM Prozessoren, die in vielen PDAs zu finden ist und auch bei dem Testgerät dieser Arbeit zum Einsatz kam. Speziell Prozessoren mit XScale-Technologie werden unterstützt, falls auf dem ausführenden System vorhanden.
Die IPP Bibliothek ist unabhängig von dem verwendeten Betriebssystem, da sie reine Rechenoperationen enthält, ohne dabei auf Betriebssystemfunktionen zurückzugreifen. Die Implementierung der FFT arbeitet auf komplexen 16 Bit vorzeichenbehafteten Ganzzahlen und kann auf Blöcken der Länge 2^n von 1 bis 4096 durchgeführt werden. Die Dokumentation der IPP lässt offen, mit welcher Wortbreite die FFT intern berechnet wird. Vermutlich ist die Genauigkeit von dem Befehlssatz des jeweiligen Prozessors abhängig. Die Ergebnisse auf einem StrongARM SA-1110 Prozessor lassen auf eine 16 Bit Ganzzahlarithmetik schließen, die für die Spracherkennung allerdings etwas zu ungenaue Ergebnisse liefert. Es konnte eine leichte Verschlechterung der Erkennungsleistung festgestellt werden. Um die schmale Wortbreite von 16 Bit besser zu nutzen, sollte deshalb das Eingabesignal vor der FFT normalisiert werden. In Tabelle 3 sind die Ergebnisse der Evaluierung festgehalten. Es fällt der extrem hohe maximale Fehler auf. Eine nähere Betrachtung der Koeffizienten ergibt, dass bei höheren Werten während der FFT viele Ganzzahl-Überläufe auftreten, die offenbar nicht abgefangen werden. Darum

Funktion	Float	IPP	IPP n: 100%	IPP n: 10%	IPP n: 2%
Zeit [ms]	622	42	46	46	46
d. Fehler [%]	0	5,8	60,2	2,4	0,7
max. Fehler [%]	0	10281,8	51431,4	2503,1	558,2
WA [%]	69,27	68,59	65,18	69,00	69,23

Tabelle 3: Benötigte Zeit, Fehler und Wortakkuratheit mit der FFT aus der IPP Bibliothek

wird nicht auf 100% des 16 Bit Wertebereichs skaliert, weil dadurch die Überläufe nur noch verstärkt werden würden. Es wurde die beste Genauigkeit ermittelt bei nur 2%. Hier finden kaum noch Überläufe statt, und die Genauigkeit der Berechnung ist noch akzeptabel. Niedrigere Werte für die Normalisierung liefern zwar einen niedrigeren maximalen Fehler, aber die Berechnung der FFT wird zu ungenau. Durch die Normalisierung auf 2% kann die Wortakkuratheit tatsächlich wieder erhöht werden. Tabelle 3 gibt einen Überblick über die Wortakkuratheit und den durchschnittlichen Fehler dieser Optimierung im Zusammenhang mit der vorangehenden Normalisierung.

3.3 Schnelle Approximation der Logarithmus-Funktion

Um den Logarithmus $\log_b f$ zur Basis b einer beliebigen Fließkommazahl f zu berechnen, benötigt ein Prozessor ohne FPU hunderte von Rechenoperationen, bis das Ergebnis genau genug iteriert ist und wieder als Fließkommazahl vorliegt.
Eine Möglichkeit, komplexere Berechnungen während der Laufzeit zu vermeiden, ist die Verwendung von Nachschlagetabellen. Durch die ungleiche Dichte des Definitionsbereichs und des Wertebereichs der Logarithmusfunktion eignet sich diese Methode hier prinzipiell nicht.
Aber durch eine geschickte Approximation, die in der Spracherkennung ausreichend ist, kann die Berechnung des Logarithmus $\log_b f$ zu einer beliebigen Basis b um ein Vielfaches beschleunigt werden.

Betrachtet wird zunächst die interne Darstellung einer 32 Bit Fließkommazahl f nach der IEEE 754 Norm, die heute in den meisten Prozessorarchitekturen und Software-Implementierungen verwendet wird. Die Zahl wird mit Vorzeichen v, normalisierter Mantisse m und Exponenten e angegeben:

	Vorzeichen	Exponent	Mantisse
Wert	v	eeeeeeee	mmmmmmmmmmmmmmmmmmmmmmm
Bit	31	30-23	22-0

Tabelle 4: Bit-Darstellung einer 32 Bit Fließkommazahl

Der eigentliche Wert der Fließkommazahl berechnet sich (bis auf Ausnahmen an Extremstellen) wie folgt:

$$f = v \cdot m \cdot 2^e$$
$$\text{wobei } 1 \leq m < 2, -127 \leq e \leq 127 \text{ und } v \in \{-1; 1\}$$

Durch einfaches, bitweises Ausmaskieren und Verschieben lassen sich die Werte von e und m bestimmen. Dabei fällt auf, dass in gewisser Weise der Logarithmus Dualis bereits durch die Angabe von e gegeben ist, wenn auch in sehr grober Näherung. Da der Logarithmus nur auf positiven Zahlen definiert ist, kann das Vorzeichen v ignoriert und als konstant 1 angenommen werden.

3.3.1 Grobe Approximation

Man betrachte nun folgende Umformung:

$$\begin{aligned} \log_b f &= \log_b(m \cdot 2^e) \\ &= \log_b m + \log_b 2^e \\ &= \log_b m + e \cdot \log_b 2 \end{aligned}$$

Approximiert man die Berechnung des Logarithmus mit $\log_b f \approx e \cdot \log_b 2$, kann man die Logarithmus-Funktion auf zwei Bitoperationen und eine Multiplikation reduzieren, wobei die Bitoperationen sehr schnell und daher vernachlässigbar sind. Der Fehler, der hierbei begangen wird, beträgt $\log_b m$. Da die Mantisse m normalisiert zwischen 1 und 2 liegt, ist der Fehler also kleiner als $\log_b 2$, was bei der Vorverarbeitung bereits ausreichend ist, wenn ein kleiner Verlust an Wortakkuratheit in Kauf genommen werden kann.

3.3.2 Lineare Approximation

Die Berechnung kann aber weiter verfeinert werden durch die Approximation des Logarithmus $\log_b m$ der noch fehlenden Mantisse auf dem Intervall $[1, 2)$. Hier bieten sich verschiedene Methoden an, die in Abbildung 4 miteinander verglichen werden.
Die einfachste Methode ist die Approximation durch eine Gerade. Daraus ergibt sich direkt:

$$\log_b m \approx (m-1)\log_b 2$$

und somit für die Fließkommazahl f:

$$\begin{aligned} \log_b f &\approx (m-1)\log_b 2 + e \cdot \log_b 2 \\ &= (m-1+e)\log_b 2 \end{aligned}$$

Die Berechnung erfordert nur zwei Additionen und eine Multiplikation.

3.3.3 Approximation mit Taylorpolynom

Eine weitere mögliche Approximation besteht durch die Taylorreihenentwicklung [Jän02] des natürlichen Logarithmus $\ln x$:

$$\begin{aligned} \ln x &= \sum_{k=1}^{\infty} (-1)^{k+1} \frac{(x-1^k)}{k} \\ &= (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - ..., 0 < x \leq 2 \end{aligned}$$

Entsprechend der geforderten Genauigkeit kann der Grad des Taylorpolynoms gewählt werden. In dieser Arbeit wurde ein Taylorpolynom zweiten

Grades gewählt, um die Berechnung einfach und schnell zu halten.

$$\begin{aligned} \ln x &\approx (x-1) - \frac{(x-1)^2}{2} \\ &= \frac{1}{2}(x-1)(3-x) \end{aligned}$$

Übertragen auf den Logarithmus $\log_b f$ einer Fließkommazahl f mit beliebiger Basis b erhält man:

$$\begin{aligned} \log_b f &= \log_b m + e \cdot \log_b 2 \\ &= \frac{\ln m}{\ln b} + e \cdot \log_b 2 \\ &\approx \frac{1}{\ln b} \cdot \frac{1}{2}(m-1)(3-m) + e \cdot \log_b 2 \\ &= \log_b 2 \left(\frac{1}{2 \ln 2}(m-1)(3-m) + e \right) \end{aligned}$$

Berechnet man die konstanten Koeffizienten $\frac{1}{2 \ln 2}$ und $\log_b 2$ vor, kommt man bei dieser Approximation des Logarithmus mit drei Additionen und drei Multiplikation aus.

3.3.4 Approximation mit verbessertem Polynom

Durch Abwandlung des Taylorpolynoms lässt sich durch Probieren ein weiteres, verbessertes Polynom zweiten Grades finden, das den Logarithmus Dualis speziell auf dem Intervall [1,2) der Mantisse m approximiert:

$$\log_2 x \approx \frac{1}{3}(x-1)(5-x)$$

Die Genauigkeit ist wesentlich höher als die des Taylorpolynoms zweiten Grades für $\ln x$, wie man Abbildung 4 entnehmen kann. Es besitzt exakt den selben Aufwand und sollte daher immer vorgezogen werden. Für die Fließkommazahl f ergibt sich daraus:

$$\begin{aligned} \log_b f &= \log_b m + e \cdot \log_b 2 \\ &= \frac{\log_2 m}{\log_2 b} + e \cdot \log_b 2 \\ &\approx \frac{1}{\log_2 b} \cdot \frac{1}{3}(m-1)(5-m) + e \cdot \log_b 2 \\ &= \log_b 2 \left(\frac{1}{3}(m-1)(5-m) + e \right) \end{aligned}$$

3.3.5 Evaluierung

Aus jeder der oben vorgestellten Varianten für die Approximation der Mantisse m wurde eine Funktion in JANUS implementiert.

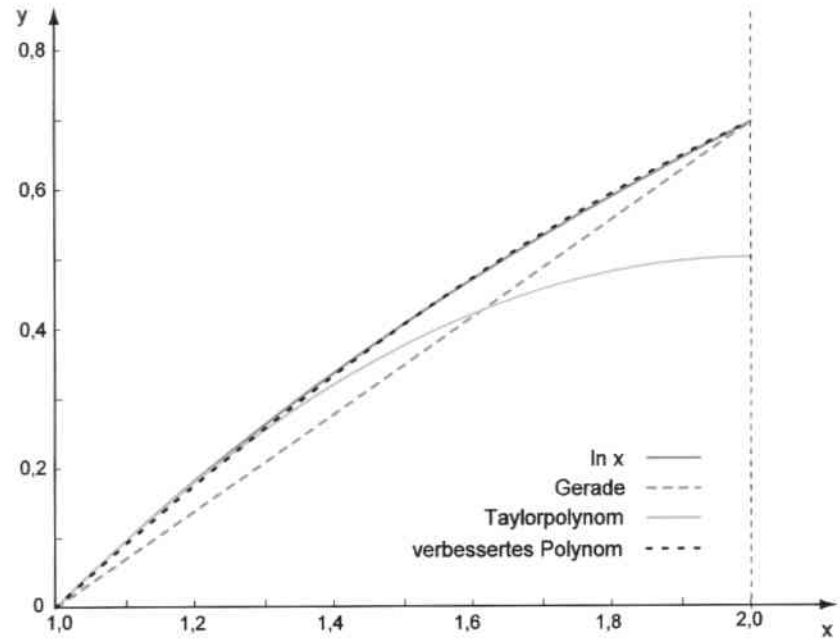

Abbildung 4: Vergleich der Approximationen für $\ln x$ auf dem Interval [1,2)

`ufastlog` (grob, nur Exponent): $\log_b f \approx \log_b 2 \cdot e$

`lfastlog` (Exp. und Gerade): $\log_b f \approx \log_b 2 \cdot (m - 1 + e)$

`tfastlog` (Exp. und Taylorp.): $\log_b f \approx \log_b 2 \cdot \left(\frac{1}{2 \ln 2}(m-1)(3-m) + e\right)$

`fastlog` (Exp. und v. Polynom): $\log_b f \approx \log_b 2 \cdot \left(\frac{1}{3}(m-1)(5-m) + e\right)$

In Tabelle 5 sind die Geschwindigkeitsgewinne gegenüber der C-Funktion `log10()` festgehalten, sowie der durchschnittliche und maximale Fehler, der während der Vorverarbeitung auf den Testdaten empirisch ermittelt wurde. Weil der absolute Zeitgewinn bei dieser Funktion nicht so groß ist, wurde in späteren Evaluierungen die Funktion `fastlog` und nicht `lfastlog` verwendet, um sicher zu gehen, dass die Worktakkuratheit so wenig wie möglich beeinträchtigt wird.

Funktion	`log10()`	`fastlog`	`tfastlog`	`lfastlog`	`ufastlog`
Zeit [ms]	114	16	16	9	5
d. Fehler [%]	0	0,03	0,32	0,30	2,61
max. Fehler [%]	0	0,05	1,44	0,44	5,22
WA [%]	69,27	69,23	69,05	69,18	68,86

Tabelle 5: Benötigte Zeit, Fehler und Einfluß auf die Wortakkuratheit

4 Optimierung der Decodierung

Die Decodierung der Merkmalsvektoren zu einer Hypothese des gesprochenen Wortlautes benötigt mit Abstand die meiste Rechenzeit während des gesamten Erkennungsvorgangs. Sie bietet daher die größte Angriffsfläche für eine Geschwindigkeits-Optimierung, stellt aber auch gleichzeitig die größte Herausforderung dar. Anders als bei der Vorverarbeitung können die Optimierungen nicht einzeln betrachtet werden, sondern müssen im Kontext der gesamten Decodierung gesehen werden. So führt etwa eine schnelle, aber ungenaue Berechnung der Mahalanobis-Distanzen zu größerer Perplexität im Suchbaum, dessen Abarbeitung sich einen Teil der gewonnen Zeit wieder zurückholt.

4.1 Das Fünf-Schichten-Modell zur Kategorisierung von Optimierungen

In [CSMR04] wird die Decodierung in vier Schichten eingeteilt, denen sich die einzelnen Optimierungen der Decodierung zuordnen lassen. Zur Vervollständigung wird in dieser Arbeit eine fünfte Schicht eingeführt, die sich auf der Ausführungsebene befindet. Optimierungen dieser Schicht bewirken im Idealfall keine Änderung des Decodierungsvorgangs, sondern lediglich eine schnellere Abarbeitung. Wird dabei auf Genauigkeit verzichtet, kann sich das Ergebnis der Decodierung aufgrund unterschiedlicher Entscheidungen im Suchbaum verändern. Das Prinzip und die Aufwandsklasse bleiben jedoch gleich. Dieser Schicht lassen sich Ganzzahlarithmetik und Nachschlagetabellen zuordnen.

Schicht	**Beispiele**
Eingabe-Schicht (Frame-Layer)	Frame Skipping, EFVR
Codebuch-Schicht (GMM-Layer)	Strahlsuche
Gaussverteilungs-Schicht (Gaussian-Layer)	BBI Tree
Komponenten-Schicht (Component-Layer)	Faules Auswerten der Distanzen
Ausführungs-Schicht (Execution-Layer)	Ganzzahlarithmetik, Nachschlagetabellen

Tabelle 6: Fünf-Schichten-Modell der Optimierungsansätze für die Decodierung

Die Motivation des Schichten-Modells ist es, die sehr verschiedenen Optimierungsansätze zur besseren Analyse zu klassifizieren. Jede dieser Schichten bietet ein Potential für Optimierungen. Darum sollte versucht werden, für

jede Schicht eine Optimierung einzusetzen.

Neben den hier vorgestellten Optimierungen wurde in dieser Arbeit die Strahlsuche zur Ermittlung der besten Hypothese des gesprochenen Textes eingesetzt. Die Strahlsuche kann als Optimierung gegenüber der vollständigen Auswertung aller Hypothesen gesehen werden, auch wenn dieses Verfahren mittlerweile ein wichtiger Bestandteil der Decodierung ist. Sie erlaubt es auch, die Geschwindigkeit der Decodierung gegen Verlust von Wortakkuratheit in hohem Maße anzupassen. Das Prinzip der Strahlsuche ist in [ST95] erläutert.
Ein weiteres Optimierungsverfahren ist der sog. „*Bucket Box Intersection Tree*" (BBI Tree). Der BBI Tree teilt den Suchraum aus Gaußverteilungen für einen Merkmalsvektor mit Hilfe einer Baumstruktur in verschiedene Bereiche ein, die sich mit zunehmender Tiefe des Baumes verfeinern. Dadurch verringert sich die Auswahl der Gaußverteilungen, die mit dem Merkmalsvektor verglichen werden müssen. Der BBI Tree wurde in dieser Arbeit mit den hier vorgestellten Optimierungen zur Steigerung der Geschwindigkeit kombiniert. Zu näheren Betrachtung sei auf [Wos98] verwiesen.

4.2 Faules Auswerten der Mahalanobis-Distanzen

Zur Vervollständigung und Vorbereitung der nächsten Optimierungen sei hier die faule Auswertung der Mahalanobis-Distanzen erwähnt.
Die meiste Zeit der Decodierung wird auf dem PDA mit der Berechnung der Mahalanobis-Distanzen eines Merkmalsvektors zu den verteilten Gaußmixturen verbracht.
Die Mahalanobis-Distanz zwischen einem Merkmalsvektor und einer Gaußverteilung ist wie folgt definiert:

$$\Delta^2 = (\vec{x} - \vec{m})\mathbf{C}^{-1}(\vec{x} - \vec{m})$$

wobei $\vec{x}$ der Merkmalsvektor, $\vec{m}$ der Mittelwertsvektor der Gaußverteilung und $\mathbf{C}$ die dazugehörige Kovarianzmatrix ist. Bei diagonalisierten Kovarianzmatrizen, wie sie bei der Decodierung in dieser Arbeit verwendet wurden, läuft die Berechnung hinaus auf:

$$\bar{\Delta}^2 = \sum_{i=1}^{n} (m_i - x_i)c_{ii}(m_i - x_i) = \sum_{i=1}^{n} (m_i - x_i)^2 c_{ii}$$

Bei der Decodierung wird immer die Gaußverteilung mit der geringsten Distanz zu dem Eingabe Vektor $\vec{x}$ gesucht. Da $\bar{\Delta}^2$ stets positiv ist und die Summe während der Berechnung monoton wächst, kann bereits nach Überschreiten des bisher gefundenen Minimums abgebrochen werden. Eine solche Optimierung wird als faule Auswertung bezeichnet (*engl.* „*Lazy Evaluation*"), da zum Vergleich mit dem bisherigen Minimum nicht immer die

komplette Summe ausgerechnet wird. Diese Optimierung verändert die Auswertung nicht und sollte deshalb immer angewandt werden. Allerdings kann das Testen der Zwischensumme länger dauern als das Addieren der nächsten Komponente. Darum sollte nicht nach jedem Summand getestet werden. Die Testintervalle hängen von den Geschwindigkeitsverhältnissen der einzelnen Befehle ab und können empirisch ermittelt werden. Bewährt haben sich kurze Intervalle am Anfang der Summe, da hier bereits die meisten Gaußverteilungen ausscheiden, und längere Intervalle am Ende, um nicht unnötig viele Vergleiche durchführen zu müssen.
Diese Optimierung lässt sich der Komponenten-Schicht zuordnen, weil nicht alle Komponenten eines Merkmalsvektors in die Auswertung miteinbezogen werden. Diese Optimierung ist in der Ausgangsbasis bereits enthalten und wird hier nicht evaluiert.

4.3 Decodierung mit Ganzzahlarithmetik

Die Mahalanobis-Distanzen werden üblicherweise mit Fließkommazahlen berechnet. Man kann also einen deutlichen Geschwindigkeitszuwachs erwarten, wenn die Berechnung auf Ganzzahlarithmetik umgestellt wird. Im Idealfall hat die Decodierung den identischen Verlauf und die Erkennungsleistung wird nicht beeinträchtigt.

4.3.1 Bestimmung der Skalierungsfaktoren

Ähnlich wie bei der Matrixmultiplikation mit Ganzzahlen müssen für eine gute Ausnutzung des Zahlenraums die Werte m_i, x_i und c_{ii} möglichst hoch skaliert werden, ohne jedoch einen Überlauf zu riskieren. Für den Merkmalsvektor $\vec{x}$ und den Mittelwertsvektor $\vec{m}$ wird der gleiche Skalierungsfaktor benötigt, da diese beiden Vektoren additiv verknüpft werden. Es ergibt sich folgende Situation:

$$({s_m}^2 s_c) \cdot \bar{\Delta}^2 \approx \sum_{i=1}^{n} (\lfloor s_m \cdot m_i \rfloor - \lfloor s_m \cdot x_i \rfloor)^2 \lfloor s_c \cdot c_{ii} \rfloor$$

Die Gaußklammern $\lfloor \ldots \rfloor$ stehen hier wieder analog zur Matrixmultiplikation für die Umwandlung von Fließkommazahlen in Ganzzahlen, wobei die Kommastellen einfach abgeschnitten werden. Anders als bei der Matrixmultiplikation werden $\vec{m}$ und $\mathbf{C}$ bereits bei der Initialisierung des Spracherkenners umgewandelt und permanent gespeichert. Während der Decodierung muss lediglich der Eingabe Vektor $\vec{x}$ einmalig skaliert werden. Dadurch ist es ohne größeren zusätzlichen Zeitaufwand möglich, Überläufe abzufangen und abzuschneiden. Gaußverteilungen mit extremen Mittelwerten oder Kovarianzen werden dadurch etwas deformiert, was aber durch einen höheren Skalierungsfaktor aller übrigen Verteilungen belohnt wird. Hier gilt es, einen

guten Kompromiss zu finden. Bei sehr kleinen Werten von c_{ii} sollte darauf geachtet werden, dass keines der $\lfloor s_c \cdot c_{ii} \rfloor$ zu Null ausgewertet wird, da sonst der Einfluß der Komponente komplett verloren geht, auch wenn die Differenz zwischen m_i und x_i sehr groß ist. In diesem Fall sollte der kleinste darstellbare, positive Wert zugewiesen werden, bei Ganzzahlen wäre das die Eins. Eine mathematisch korrekte Abschätzung für s_m und s_c zu finden, ist unmöglich, da die Merkmalsvektoren $\vec{x}$ während der Initialisierungsphase nicht zur Verfügung stehen. Die x_i benötigen aber den gleichen Skalierungsfaktor wie die m_i, die auf jeden Fall im Voraus berechnet werden müssen, weil die Skalierung der kompletten Codebücher während der Laufzeit natürlich nicht sinnvoll ist.
Im Folgenden sei zur textuellen Vereinfachung:

$$\begin{aligned} \dot{m}_i &= s_m \cdot m_i \\ \dot{x}_i &= s_m \cdot x_i \\ \dot{c}_{ii} &= s_c \cdot c_{ii} \end{aligned}$$

Bei dieser Arbeit wird für die Ganzzahlen eine Wortbreite von 32 Bit eingesetzt. Da die Distanz Δ^2 stets positiv ist, kann unter Verwendung von vorzeichenlosen Ganzzahlen die volle Wortbreite für die Summation genutzt werden. Wir versuchen nun s_m und s_c möglichst groß zu wählen. In den Tests hat sich gezeigt, dass nicht zu viele Gaußverteilungen deformiert werden dürfen. Es sollte daher für mehr als 90% der Gaußverteilungen erfüllt sein:

$$-2^{10} < \dot{m}_i < 2^{10} \tag{3}$$

$$0 < \dot{c}_{ii} < 2^8 \tag{4}$$

Die Komponenten des Merkmalsvektors $\vec{x}$ bekommen den gleichen Wertebereich wie die Mittelwertsvektoren zugewiesen. Überschreitungen der Grenzen werden abgeschnitten. Für die $\dot{c}_{ii}$ wird ein etwas kleinerer Wertebereich gewählt, da die Kovarianzen laut [VISV04] gegenüber Fehlern durch Quantisierung weniger anfällig sind. Die Bestimmung der Skalierungsfaktoren kann während der Initialisierung des Decoders anhand eines Histogramms der m_i und c_{ii} über alle Codebücher vorgenommen werden. Wird das akustische Modell als gegeben vorausgesetzt, können auch die Faktoren fest vorgegeben werden.
Es ergibt sich im schlechtesten Fall:

$$\begin{aligned} |(\dot{m}_i - \dot{x}_i)| &< 2^{11} \\ \Rightarrow (\dot{m}_i - \dot{x}_i)^2 &< 2^{22} \\ \Rightarrow (\dot{m}_i - \dot{x}_i)^2 \dot{c}_{ii} &< 2^{30} \end{aligned}$$

Ein Summand von $\bar{\Delta}^2$ benötigt also im schlechtesten Fall 30 Bit. Ist die bisherige Zwischensumme kleiner als 2^{30}, dann können noch sechs Summanden addiert werden, bevor ein Überlauf auftreten kann. Da diese Methode

aber mit der faulen Auswertung kombiniert wurde, die maximal nach vier Summanden testet, ob das bisherige Minimum von $\bar{\Delta}^2$ nicht überschritten wurde, und als initiales Minimum $2^{30} - 1$ gewählt wird, kann kein Überlauf auftreten.

4.3.2 Verbesserte Genauigkeit durch Bit-Schieben

Die Genauigkeit der Berechnung der Mahalanobis-Distanzen kann verbessert werden, wenn man den Wertebereich der Mittelwerte und Kovarianzen höher wählt als eigentlich zulässig. Während der Berechnung eines Summanden von $\bar{\Delta}^2$ wird dann mit Hilfe einer Bit-Schiebe-Operation das Zwischenergebnis auf den zulässigen Wertebereich verringert. Im Folgenden sei „$\gg_b$" eine vorzeichenerhaltende Bit-Verschiebung um b Stellen nach rechts, was bei positiven Ganzzahlen einer Ganzzahldivision durch 2^b entspricht. Es ergibt sich folgende Berechnung:

$$\frac{({s_m}^2 s_c)}{2^{b_1+b_2}} \cdot \bar{\Delta}^2 \approx \sum_{i=1}^{n} \Big((\lfloor s_m \cdot m_i \rfloor - \lfloor s_m \cdot x_i \rfloor)^2 \gg_{b_1} \Big) \lfloor s_c \cdot c_{ii} \rfloor \gg_{b_2}$$

Die Bitoperationen verursachen keine zusätzliche Rechenzeit, da der Prozessor auf den langsamen Speicher warten muss und in der Zwischenzeit diese Operationen „kostenlos" ausführen kann. Der Skalierungsfaktor s_m und die Grenze für die Mittelwerte kann jetzt um den Faktor $\sqrt{2^{b_1}}$ erhöht werden, und s_c die Kovarianzen mit ihren Grenzen um den Faktor 2^{b_2}. Allerdings muss darauf geachtet werden, dass der Ausdruck vor den Bitoperationen innerhalb der Grenzen des 32 Bit Ganzzahlraumes bleibt.
Da die zusätzlich gewonnenen Binärstellen durch das Bit-Schieben sukzessive wieder abgebaut werden, wirkt sich die genauere Berechnung von $(\lfloor s_m \cdot m_i \rfloor - \lfloor s_m \cdot x_i \rfloor)^2$ und der Multiplikation mit $\lfloor s_c \cdot c_{ii} \rfloor$ nur wie eine Rundung auf das Endergebnis der Mahalanobis-Distanz aus.

4.3.3 Beschleunigung durch serialisierte Speicherzugriffe

Für die Berechnung der Mahalanobis-Distanzen wird immer abwechselnd ein Mittelwert und seine dazugehörige Kovarianz der Gaußverteilung aus dem Speicher gelesen. Für Systeme, die Speicherinhalte in Reihe schneller lesen als willkürliche Zugriffe, können die Mittelwerte und Kovarianzen verschachtelt im Speicher abgelegt werden. In dieser Arbeit wurden die Mittelwerte und Kovarianzen zusätzlich mit einer verringerten Wortbreite von 16 Bit gespeichert, da die Beträge beider Werte nach (3) und (4) immer kleiner als 2^{10} sind. Eine starke Beschleunigung kann aber nicht erwartet werden, da maximal nur 32 Wertepaare in Reihe gelesen werden, wenn die Anzahl der Dimensionen des Merkmalsvektors 32 beträgt. Zudem ist wegen des 32 Bit breiten Datenbusses ein 16 Bit Speicherzugriff nicht schneller als ein 32 Bit Speicherzugriff.

4.3.4 Evaluierung

Bei der Evaluierung wurden neben der Ausgangsbasis drei Varianten der Decodierung mit Ganzzahlarithmetik eingesetzt:

float	: Decodierung mit Fließkommaarithmetik
Int (128, 64)	: Decodierung mit Ganzzahlarithmetik $s_m = 128$; $s_c = 64$
Int_b (4096, 256)	: verbesserte Genauigkeit durch Bit-Schieben $s_m = 4096$; $s_c = 256$; $b_1 = 2^{10}$; $b_2 = 2^2$
Int_s (128, 64)	: Beschleunigung durch serialisierte Speicherzugriffe $s_m = 128$; $s_c = 64$

Als Zeit wurde die Verweildauer in der Funktion für die Berechnung der Mahalanobis-Distanzen gemessen. Die Funktion mit verbesserter Genauigkeit liefert eine geringfügig schlechtere Wortakkuratheit, die anderen beiden Varianten eine bessere Wortakkuratheit als die Ausgangsbasis. Solche geringen Abweichungen sind zufällig bedingt, da die Decodierung aufgrund der leicht abweichenden Ergebnisse für Δ^2 unterschiedlich abläuft. Als Schlussfolgerung kann man annehmen, dass eine höhere Skalierung, als sie bereits bei Int und Int_s verwendet wurde, zwar leichte Unterschiede bewirkt, aber nicht mehr zur Verbesserung der Wortakkuratheit beiträgt.
An der sogar etwas schnelleren Berechnung von Int_b im Vergleich zu Int kann abgelesen werden, dass die Bit-Schiebe-Operationen keine zusätzliche Rechenzeit benötigen, weil der Prozessor ohnehin auf den nächsten Speicherzugriff warten muss.
Allerdings sollte hier beachtet werden, dass die Meßgenauigkeit für die Zeit bei der Bewertungsfunktion besonders ungenau ist, da sie für jeden Merkmalsvektor einzeln aufgerufen wird. Bei jedem Aufruf kann ein Meßfehler von bis zu 1ms entstehen. Des Weiteren wurden Schwankungen in der Rechenleistung des PDA beobachtet, die besonders bei Int_s aufgefallen sind. Darum wurde während der abschließenden Evaluierung mit Int gearbeitet, und nur am Ende ein Vergleich mit Int_s durchgeführt. Da aber Int_s und Int exakt gleich ablaufen, ist kein Nachteil durch die Verwendung von Int_s zu erwarten.

Funktion	float	Int	Int_b	Int_s
Zeit [ms]	59.989	6.102	5.739	5.203
WA [%]	69,27	69,32	69,05	69,32

Tabelle 7: Einfluß der Ganzzahlarithmetik auf die Wortakkuratheit

4.4 Decodierung mit Nachschlagetabellen

Eine Alternative zur Decodierung mit Ganzzahlarithmetik stellt die Verwendung von Nachschlagetabellen dar, wie sie in [VISV04] beschrieben ist. Mit Hilfe dieser Tabellen wird die Berechnung eines Summanden der Mahalanobis-Distanz auf einen Speicherzugriff reduziert.

4.4.1 Temporäre Nachschlagetabellen

	0	1	$\cdots$	$q_m - 1$
0				
1				
$\vdots$				
$q_c - 1$				

Tabelle 8: Nachschlagetabelle für eine Dimension des Merkmalsvektors

Zuerst müssen alle Codebücher der Akustik quantisiert werden. Als Quantisierungsstufen werden q_m Stufen für die Mittelwerte und q_c Stufen für die Kovarianzen gewählt. Zusätzlich müssen noch m_{min}, m_{max}, c_{min} und c_{max} bestimmt werden, um die Grenzen des Wertebereichs anzugeben. Hier können ähnliche Grenzen wie bei der Decodierung mit Ganzzahlarithmetik verwendet werden. Durch Tests wurde ermittelt, dass etwa 90% der Gausverteilungen nicht abgeschnitten werden sollten. Danach kann mit der Quantisierung der Codebücher begonnen werden. Jede Komponente einer Gaußverteilung, bestehend aus Mittelwert m_i und Kovarianz c_{ii}, wird quantisiert und einer Zelle der zweidimensionalen Tabelle zugeordnet. In Tabelle 8 ist eine solche Nachschlagetabelle für eine Komponente eines Merkmalsvektors skizziert. Um nicht während des Zugriffs auf eine Zelle die beiden Indizes für Mittelwert und Kovarianz zu der Adresse berechnen zu müssen, kann statt der beiden Indizes im Codebuch auch direkt die Adresse eingetragen werden. Hierfür genügt ein 16 Bit Wert, wenn die Tabellen auf 256×256 Einträge beschränken werden, oder ein 32 Bit Wert, wenn direkt der Zeiger im Speicher auf die Zelle im Codebuch abgelegt wird. Danach ist die Initialisierungphase abgeschlossen.
Jetzt geht es in die Decodierungsphase. Man erhält einen Merkmalsvektor als Eingabe. Für jede Komponente des Merkmalsvektors werden alle Mahalanobis-Distanzen zu jedem Repräsentanten einer Zelle der Nachschlagetabelle berechnet. Man benötigt also genauso viele Tabellen wie Dimensionen des Merkmalsvektors. Auf den ersten Blick sieht diese Berechnung mit Aufwand $O(n^3)$ während der Decodierung recht aufwendig aus. Die Distanzen können aber bei linearer Quantisierung in äquidistanten Schritten berechnet werden, wodurch die Implementierung wesentlich effizienter gestaltet werden kann als bei willkürlichen Eingaben, wie sie bei der eigentlichen

Decodierung anfallen würden. Zusätzlich kann zur Beschleunigung für die Berechnung der Tabellen Ganzzahlarithmetik verwendet werden.
Der Index aus dem quantisierten Codebuch zeigt jetzt auf die Zelle in der Nachschlagetabelle, die einen Summanden der Mahalanois-Distanz der jeweiligen Komponente enthält.

4.4.2 Globale Nachschlagetabellen

Der Zeitverlust durch das Erstellen der temporären Tabellen kann vermieden werden, indem auch der Merkmalsvektor $\vec{x}$ quantisiert wird und eine einzige, drei dimensionale Tabelle angelegt wird. Auf der dritten Achse wird der Wert von x_i angetragen und die Tabelle während der Initialisierung des Decoders ausgefüllt. Der Merkmalsvektor wird dann während der Decodierungsphase in Tabellen-Indizes entlang dieser Achse transformiert. Die Quantisierungsparameter können zwar frei gewählt werden, jedoch bringt nach [VISV04] eine feinere Quantisierung als die Mittelwerte mit q_m keinen großen Vorteil mehr. Die drei dimensionale Tabelle besitzt also die Größe $q_m \times q_c \times q_m$. Ziel ist es, die Tabellen möglichst klein zu halten, um den Daten-Cache des PDAs möglichst gut auszunutzen.

4.4.3 Evaluierung

Funktion	float	quant2 (24, 16)	quant2 (19, 13)
Zeit [ms]	59.989	15.555	9.373
WA [%]	69,27	67,55	64,73

Tabelle 9: Benötigte Zeit und Einfluß auf die Wortakkuratheit

Es wurde nur die Quantisierung mit globalen Nachschlagetabellen evaluiert, da sie etwas schneller sind, und die temporären Tabellen eigentlich nur eine Vorüberlegung darstellen. Als Größe der Tabellen wurde für die Mittelwerte und Kovarianzen $q_m = 24$ und $q_c = 16$ gewählt. Für ein weiteres Experiment wurde eine kleinere Tabelle mit $q_m = 19$ und $q_c = 13$ gewählt, um den Geschwindigkeitsgewinn festzuhalten. Aus Tabelle 9 kann entnommen werden, dass diese Art der Beschleunigung wesentlich schlechter abschneidet als die Ganzzahlarithmetik, weil der Zugriff auf die Tabelle eine zusätzlichen Speicherzugriff erfordert. Dieser Speicherzugriff springt innerhalb der Tabellen willkürlich umher. Diese Eigenschaft ist denkbar ungünstig für den kleinen Daten-Cache des PDAs, der bei dem Testgerät nur 8KB groß ist. Darum sinkt die Geschwindigkeit dieser Methode drastisch mit zunehmender Größe der Tabellen. Um eine Beschleunigung zu erreichen, die vergleichbar mit der Decodierung mit Ganzzahlarithmetik ist, müssen die Tabellen sehr klein gewählt werden und damit die Quantisierung sehr grob. Darunter leidet die Wortakkuratheit deutlich.

Da diese Optimierung der Decodierung weder in Geschwindigkeit noch in Wortakkuratheit mit der Ganzzahlarithmetik konkurrieren kann, wurde sie in dieser Arbeit und speziell in der abschließenden Evaluierung nicht weiter verfolgt. Allerdings wurde in dieser Arbeit eine lineare Quantisierung angewandt. Es kann nach [VISV04] eine Verbesserung der Wortakkuratheit bei gleicher Größe der Tabellen erzielt werden, wenn eine Quantisierung angewandt wird, die der Verteilung der Mittelwerte und der Kovarianzen besser entspricht. Allerdings bremsen auch in diesem Fall die Speicherzugriffe den Prozessor aus.

4.5 Frühe Reduzierung der Merkmalsvektoren (EFVR)

Die frühe Reduzierung der Merkmalsvektoren (*engl.* ***E**arly **F**eature **V**ector **R**eduction*, EFVR) ist eine Abwandlung des bedingten Überspringens von Zeitfenstern (*engl. Conditional Frame Skipping*), wie es in [Wos98] beschrieben ist. Diese Optimierung ist auf der Eingabe-Schicht angesiedelt und kann daher mit den anderen hier vorgestellten Optimierungen effizient kombiniert werden.
Die Idee basiert auf der Beobachtung, dass viele Merkmalsvektoren zu einigen, wenigen Phonemen decodiert werden. Daher lassen die Merkmalsvektoren eine hohe Redundanz untereinander vermuten. Anstatt die teure Decodierung diese Redundanz abarbeiten zu lassen, wird nach einer schnelleren Möglichkeit gesucht, um dies vorher zu tun. Die einfachste Methode, das sog. „*Downsampling*", verwirft einfach jedes zweite Zeitfenster, noch bevor es von der Vorverarbeitung zu einem Merkmalsvektor aufbereitet worden ist. Möglicherweise gehen aber wertvolle, kurze Schallereignisse verloren. Daher werden zunächst alle Merkmalsvektoren berechnet, und anschließend zwei aufeinander folgende Vektoren $\vec{x}$ und $\vec{y}$ mit einem Distanzmaß verglichen. Ein Merkmalsvektor wird nur dann übersprungen, wenn die Distanz zum Vorgänger einen bestimmten Schwellwert unterschreitet. In [Wos98] wird dazu die Euklidsche Distanz verwendet und dem übersprungenen Vektor wird der Wert der Bewertungsfunktion seines Vorgängers zugeordnet. In dieser Arbeit wird die Euklidsche Distanz leicht abgewandelt:

$$\delta^2_{\vec{x},\vec{y}} = \sum_{i=1}^{n} \frac{(x_i - y_i)^2}{i}$$

Dadurch wird der Anteil der höheren Koeffizienten an der Summe für die Distanz verringert. Es wird ausgenutzt, dass die Koeffizienten der Merkmalsvektoren nach der LDA so sortiert sind, dass der Informationsgehalt über die Klassenzugehörigkeit eines Merkmalsvektors mit zunehmenden Index abnimmt.
Der Unterschied zum *Conditional Frame Skipping* besteht nun in der Überlegung, dass die Informationen des übersprungenen Vektors nicht komplett verloren gehen sollen. Also wird er zu seinem Vorgänger addiert und das

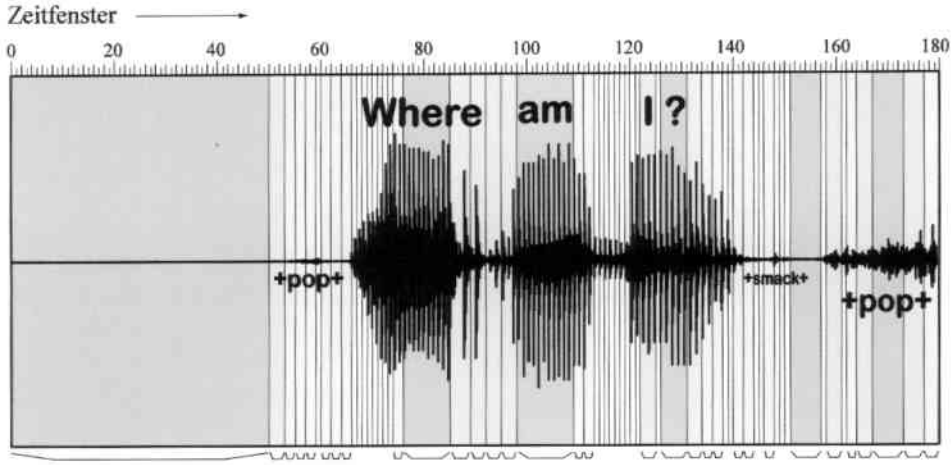

Abbildung 5: EFVR auf dem Satz „Where am I ?“

arithmetische Mittel der beiden Vektoren gebildet. So können auch mehrere aufeinander folgende Merkmalsvektoren zusammengefasst werden, wobei jeder Vektor das gleiche Gewicht erhält.
In Abbildung 5 ist eine Äußerung dargestellt, die ursprünglich in 180 Zeitfenster unterteilt worden ist und daher 180 Merkmalsvektoren für die Decodierung liefert. Ähnliche, aufeinander folgende Vektoren wurden zu nur 60 Vektoren zusammengefasst, die als einfarbige Blöcke eingezeichnet sind. Deutlich ist zu sehen, wie stille Passagen und die drei längeren Vokale zusammengefasst werden. Im Durchschnitt ließ sich auf den Testdaten ohne Verlust an Wortakkuratheit die Anzahl der Vektoren während der Sprache etwa um 25% reduzieren. Bezieht man kurze Sprechpausen mit ein, wie sie in der Datenbasis vorhanden waren, können die Merkmalsvektoren um etwa 40% reduziert werden.
Erstaunlicherweise ist bei Einsatz dieser Optimierung sogar eine leichte Verbesserung der Wortakkuratheit zu beobachten. Bei Sprechpausen wird hauptsächlich Rechenzeit eingespart, weil weniger Vektoren zu decodieren sind. Interessant sind aber die Vokale. Da hier über mehrere, ähnliche Merkmalsvektoren gemittelt wird, befindet sich der resultierende Vektor in der Mittel des Clusters, und kleine Abweichungen durch Rechenungenauigkeit oder Fluktuation in der Stimme werden kompensiert. Die Suche kann den Vektor mit einer höheren Bewertung der Klasse zuordnen. Wenn n benachbarte Vektoren zusammengefasst worden sind, wird seine Bewertung nicht n-mal hintereinander für die ausgelassenen Vektoren verwendet, sondern das Cluster wird nur als ein einziger Vektor behandelt, mit einer n-mal höheren Bewertung. Dadurch sinkt die Perplexität im Suchbaum, und nicht nur die Berechnung der Distanzen, sondern auch die Suche wird beschleunigt.

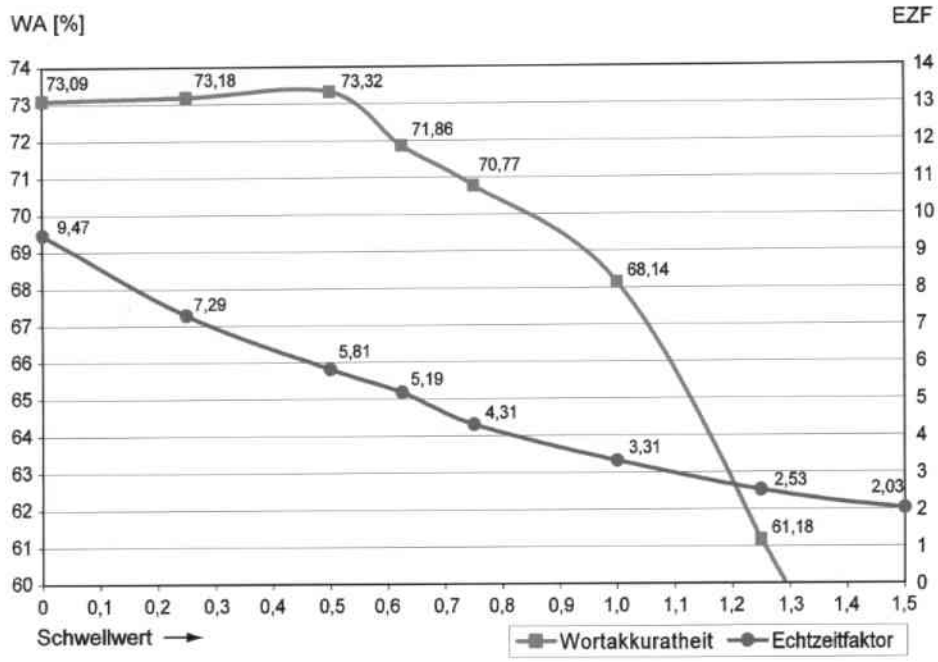

Abbildung 6: Vergleich des Echtzeitfaktors und Wortakkuratheit in Abhängigkeit des Schwellwertes der EFVR

4.5.1 Evaluierung

Die Evaluierung wurde bereits in Kombination mit optimierter Vorverarbeitung und Decodierung mit Ganzzahlarithmetik durchgeführt. Der Einfluss auf den Decodierungsvorgang lässt sich aber dennoch gut ablesen, da sich diese Optimierungen kaum beeinflussen. Die Evaluierung wurde bereits mit der OFS-Akustik durchgeführt, die auch in der abschließenden Evaluierung verwendet wird. Für eine Beschreibung sei auf Kapitel 6 verwiesen.
In Tabelle 10 ist eine Testreihe festgehalten, die den Schwellwert in kleinen Schritten erhöht. Als Ausgangsbasis ist hier der Schwellwert 0 zu sehen, da hier keine Vektoren zusammengefasst wurden. Es ist die Anzahl der Vektoren eingetragen, die beginnend mit 100% ständig abnimmt, da bei höherem Schwellwert mehr Vektoren zusammengefasst werden. Zusätzlich ist die Zeit eingetragen, die für die gesamte Decodierung ohne Vorverarbeitung benötigt wurde.
In Abbildung 6 ist die Testreihe noch einmal grafisch dargestellt. Die Berechnung des Echtzeitfaktors beinhaltet die Vorverarbeitung nicht. Deutlich zu sehen ist ein Einbruch der Wortakkuratheit ab einem Schwellwert von 0,625, da hier bereits Merkmalsvektoren zusammengefasst werden, die zu verschiedenen Merkmalsklassen gehören. Bei genauer Betrachtung der Wortakkuratheit steigt die Anzahl der ausgelassenen Wörter (DEL), während

die falsch eingefügten Wörter (INS) zurückgehen.

Schwellwert	0	0,25	0,5	0,625	0,75	1,0	1,25	1,5
Vektoren [%]	100,0	71,2	59,1	55,3	50,2	42,8	36,9	31,1
Zeit [ms]	9.472	7.293	5.808	5.192	4.310	3.311	2.534	2.033
DEL [%]	7,1	7,2	8,4	9,7	11,1	13,5	17,9	24,7
INS [%]	3,6	3,4	2,4	2,1	2,1	1,7	1,2	0,9
WA [%]	73,09	73,18	73,32	71,86	70,77	68,14	61,18	51,91

Tabelle 10: Zeit und Wortakkuratheit der EFVR in Abhängigkeit des Schwellwertes der EFVR

5 Implementierung

Dieses Kapitel beschäftigt sich mit der Portierung des Spracherkenners JANUS auf die Windows CE 3.0 Plattform und den Problemen, die bei der Implementierung der Optimierungen aufgetreten sind.
Alle Optimierungen wurden so implementiert, dass sie sich auch mit anderen Konfigurationen des Spracherkenners JANUS kombinieren lassen. So erwartet beispielsweise die optimierte Matrixmultiplikation eine Fließkommazahlen-Matrix der Klasse `FMatrix` als Eingabe und besitzt auch eine solche als Ausgabe, obwohl sie intern mit Ganzzahlarithmetik arbeitet. Die Optimierung kann daher als eine Black Box mit der gleichen Schnittstelle wie die Standardfunktion gesehen werden. Dadurch können die Optimierungen auch ohne Änderung des Kontextes einzeln aktiviert oder deaktiviert werden. Im Anhang A sind alle in dieser Arbeit implementierten Optimierungen und Hilfsfunktionen dokumentiert.
Zunächst wird die verwendete Arbeitsumgebung beschrieben.

5.1 Rahmenbedingungen des praktischen Teils

5.1.1 Verwendete Hard- und Software

In dieser Arbeit wurde zum Entwickeln und Evaluieren ein Compaq iPAQ H3600 mit dem Betriebssystem PocketPC 2002 ($\hat{=}$ Microsoft Windows CE 3.0) verwendet. Dieser PDA besitzt einen Intel StrongARM SA-1110 Mikroprozessor mit 206MHz und 64MB SDRAM. Die 32 Bit breite Schnittstelle zum Arbeitsspeicher arbeitet im sog. BurstMode, der immer 16 Bytes auf einmal in den 8KB großen Daten-Cache lädt. Dadurch sind sequenzielle Speicherzugriffe etwas schneller als willkürliche Speicherzugriffe.

Als Host Rechner wurde ein Laptop mit Microsoft Windows XP verwendet. Als Entwicklungsumgebung kam Microsoft eMbedded Visual C/C++ 3.0 zum Einsatz, das derzeit frei erhältlich ist. Das SDK für PocketPC 2002 enthält eine Emulation eines PDAs oder Smartphone, die als Ersatz für einen echten PDA dienen soll. Die Emulation ist aber nicht auf Binär-Ebene kompatibel und sollte nicht für Geschwindigkeitsmessungen oder Bewertung der tatsächlichen Funktionstüchtigkeit eines Projekts herangezogen werden. Der Test auf einem echten PDA ist in jedem Fall erforderlich.

5.1.2 Limitierungen der Hardware

Die 64MB Arbeitsspeicher, die der PDA von Hause aus mit sich bringt, müssen zwischen den laufenden Anwendungen und dem virtuellen Laufwerk für Dateien dynamisch aufgeteilt werden. Das virtuelle Laufwerk kann über die üblichen Datei-Befehle in der Programmiersprache C/C++ angesprochen werden. Dateien werden transparent für den Benutzer mit einem verlustfreien

Packverfahren gesichert. Zur Datenkompression ist es also nicht erforderlich, Codebücher, Wörterbücher oder Audio-Dateien mit einem solchen Verfahren zu packen.

5.1.3 Limitierungen des Betriebssystems

Den engsten Flaschenhals unter Windows CE 3.0 stellt die Limitierung des Speichers auf 32MB für eine laufenden Prozess dar, auch wenn der Arbeitsspeicher des PDAs noch lange nicht ausgelastet ist. Diese 32MB müssen sich die ausführbare Datei, der Stapelspeicher und die reservierten Speicherblöcke teilen. Der Stapelspeicher muss fest angegeben werden und kann sich nicht dynamisch während der Laufzeit den Anforderungen anpassen. Zusätzlich fehlt die Möglichkeit, Arbeitsspeicher auf einen Massenspeicher auszulagern.
Ein weiteres großes Problem, mit dem man während der Evaluierung konfrontiert wird, ist die Fragmentierung des Arbeitsspeichers. Die Fragmentierung findet innerhalb der festen 32MB statt und kann nicht rückgängig gemacht werden. So kann beispielsweise die Situation auftreten, dass die Anwendung nur 24MB belegt, der PDA sogar noch insgesamt 32MB frei hat, aber eine Speicherreservierung von 128KB schlägt bereits fehl, weil sich kein zusammenhängender Block im virtuellen Adressraum mehr finden lässt.
Um das Betriebssystem klein zu halten, wurden viele Funktionen der Windows API nicht unter Windows CE implementiert. Zur Portierung von JANUS fehlten vor allem einige Zeit-Funktionen, Zeichenketten-Funktionen und alle POSIX[6] Funktionen. Eine Eingabe Konsole ist ebenso wenig zu finden wie die Möglichkeit, einer Anwendung Startparameter auf dem üblichen Weg zu übergeben.
Des Weitern gibt es unter Windows CE nicht das Konzept der relativen Pfade und des aktuellen Verzeichnisses. Daher müssen alle Pfade in den TCL Skripten absolut angeben werden. Der Name des initialen TCL Skriptes wird dann fest vergeben und die komplette Umgebung von JANUS muss sich in einem festgelegten Verzeichnis befinden.

5.2 Portierung von JANUS auf WinCE 3.0

Bevor der Quelltext kompiliert werden konnte, mussten für viele der fehlenden, aber benötigten Funktionen ein Workaround geschrieben werden, oder eine alternative C-Funktion mit Hilfe von Makros überlagert werden. Es wurde eine Zeitmessung integriert, da unter Microsoft eMbedded Visual C/C++ 3.0 kein Profiler für Funktionen zur Verfügung steht.
Alle Ausgaben wurden in eine Log-Datei umgeleitet, um sie später auswerten

[6]POSIX Funktionen wurden im IEEE 1003.1 (POSIX.1) Standard für eine Schnittstelle zum Dateisystem definiert, die unabhängig von der Programmiersprache und dem Betriebssystem ist. Beispiele sind open(), read(), write() und close().

zu können. Die berechneten Hypothesen werden in eine zweite Log-Datei gespeichert, die anschließend auf Wortakkuratheit geprüft werden kann. Diese Datei dient JANUS auch dazu, bei einer Unterbrechung der Evaluierung die bereits ausgewerteten Äußerungen zu überspringen.
Um den Speicherverbrauch auch temporär möglichst gering zu halten und eine zu starke Fragmentierung zu vermeiden, wurden für den Suchbaum und die Grammatik ein sog. Dump erstellt, der bereits die fertig aufgebaute Struktur enthält, und nicht Knoten für Knoten in kleinen Speicherblöcken angelegt werden muss.
Die TCL Skripte mussten so umgeschrieben werden, dass alle Pfadangaben einen absoluten Pfad im Dateisystem des PDAs darstellen. Das initiale TCL Skript benötigt den vordefinierten Namen „desc.txt", da dieser Dateiname mangels Startparameter fest in JANUS implementiert werden musste.
Es gibt eine Portierung der Skriptsprache TCL für Windows CE, jedoch musste der Spracherkenner möglichst klein gehalten werden. Daher wurde mit dem integrierten *TCL Wrapper* von JANUS gearbeitet, der eine Teilmenge der Funktionalität der TCL Skriptsprache enthält. Schleifen sind damit allerdings nicht möglich, daher wurde eine zusätzliche Funktion `loop:` in Kombination mit `scanADC` implementiert, um die Liste der Audio-Dateien automatisiert abarbeiten zu können.
Um den Quelltext von JANUS für die Windows CE 3.0 Plattform mit Microsoft eMbedded Visual C/C++ 3.0 compilieren zu können, sind folgende Angaben in Form von Compiler-Flags notwendig:
`ARM, _ARM, UNDER_CE=$(CEVersion), _WIN32_WCE=$(CEVersion)`
`$CePlatform, UNICODE, _UNICODE`
Zusätzlich müssen folgende JANUS spezifische Flags angegeben werden:
`DISABLE_TCLTK, WINDOWS, WINCE, NO_NET, IBIS`
Für die Einbindung der *Intel Performance Primitives* muss die Bibliothek „ippSP_XSC40PPC_r.lib" für den Linker angegeben werden. Die Include-Dateien der IPP müssen durch Angabe des Include-Datei-Pfades dem Compiler zur Verfügung gestellt werden. JANUS benötigt unter Windows CE eine feste Größe von 0x80000 Bytes (≙512KB) für den Stapelspeicher.

5.3 Probleme bei der Implementierung und Evaluierung

Trotz der oben genannten Anpassungen sind noch einige Probleme aufgetreten. So besitzt eMbedded Visual C/C++ 3.0 einen Compiler-Fehler, der sich erst bemerkbar macht, wenn man eine ausführbare Datei ohne Debug-Informationen erstellt. An einer bestimmten Stelle des Quellcodes wurden Fließkommazahlen immer zu 0 ausgewertet. Bei eingeschaltetem Debugger wurde die betreffende Berechnung korrekt ausgewertet. Ein Workaround wurde durch Vermeidung von Fließkommazahlen an der betreffenden Stelle implementiert. Zusätzlich weicht die Berechnung der Fließkommazahlen mit und ohne Debug Informationen geringfügig voneinander ab, was auf die Ein-

stellungen der Optimierungstufe des C-Compilers zurückzuführen ist.
Zur Evaluierung wurden 363 Sprachäußerungen herangezogen, die zusammen ca. 15min Tonaufnahme darstellen. Diese belegen bei einer Abtastrate von 16kHz und 16 Bit Auflösung ungefähr 26,7MB Speicherplatz. Zusammen mit dem Sprachmodell (8MB), dem BBI Tree (6MB), der Grammatik (1,4MB) und der ausführbaren Datei von JANUS (1,5MB) ergibt sich ein zu hoher Speicherverbrauch, da mindestens 32MB der 64MB Arbeitsspeicher für den Spracherkenner während der Laufzeit übrig sein sollten. Daher wurde während der Entwicklung mit 8kHz Abtastrate gearbeitet. Die abschließende Evaluierung mit 16kHz Abtastrate wurde dann in zwei Schritten jeweils auf der halben Datenmenge durchgeführt. Für weitere Studien sollte mit einer zusätzlichen Speicherkarte gearbeitet werden, um den Evaluierungsprozess nicht unterbrechen zu müssen.
Letztendlich bleibt aber noch das Problem der Speicherfragmentierung. Gerade während der Decodierung werden viele kleine Speicherblöcke belegt und wieder freigegeben, die zu einer starken Fragmentierung des Speichers führen. Die Folge ist ein Abbruch der Evaluierung wegen Speicherknappheit. Der Spracherkenner muss danach neu gestartet werden und bei der zuletzt evaluierten Äusserung fortfahren. Dieses Problem trat besonders bei Evaluierungen mit großer Strahlbreite und dem BBI Tree auf, der zusätzlich etwa 6MB benötigt. Aus diesem Grund wurde auch auf eine Sprecher- und Kanaladaption verzichtet, da die Ergebnisse durch die Unterbrechung nicht reproduzierbar gewesen wären. Dadurch verringert sich die absolute Wortakkuratheit der Ergebnisse in dieser Arbeit. In Tabelle 11 ist ein Vergleich der Wortakkuratheit mit der Ausgangsbasis mit Adaption festgehalten, bei der sowohl Training als auch die Tests mit Adaption durchgeführt wurden. Es wurden die akustischen Modelle OFS und MS betrachtet, die in der abschließenden Evaluierung verwendet wurden und dort beschrieben sind.
Ein Ziel der Optimierung des Speicherbedarfs von JANUS sollte daher die Vermeidung der Fragmentierung des Speichers sein, etwa durch ein Block-Speicher-Management, wie es in Teilen von JANUS bereits implementiert ist.

WA	ohne Adaption	mit Adaption
OFS	73,77%	76,82%
MS	73,36%	76,82%
OFS (BBI Tree)	73,64%	76,86%
MS (BBI Tree)	72,18%	76,00%

Tabelle 11: Vergleich der Wortakkuratheit ohne und mit Adaption

6 Abschließende Evaluierung

Die abschließende Evaluierung zeigt die effizientesten Optimierungen im Zusammenspiel. Es wurde hierfür der gleiche Datensatz benutzt wie bei den Tests für die einzelnen Optimierungen. Es wurden jedoch zwei weitere, verschiedene akustische Modelle mit gleicher Größe verwendet.
Jedes Modell besitzt 1998 Codebücher mit 16 Gaußverteilungen auf einem 32 dimensionalen, MFCC[7] basierenden Merkmalsraum. Die Trainingsdaten wurden alle in 16kHz und 16 Bit Auflösung aufgenommen und bestanden aus 276h Nahbesprechungsaufnahmen von Meetings aus verschiedenen Quellen [MJF+04]. Im Training wurde die VTLN eingesetzt, wobei der Warp-Faktor über die gesamten Trainingsdaten gemittelt wurde und nicht, wie sonst üblich, sprecherabhängig bestimmt wurde. Der Grund hierfür ist, dass die VTLN in die Vorverarbeitung mit aufgenommen werden sollte, damit sie in die Evaluierung mit einfließt und später zur Verfügung stehen kann.
Die Testdaten bestanden aus 363 Aufnahmen in 16kHz und 16 Bit Auflösung, die mit einem Laptop und einem Nahbesprechungsmikrophon aufgezeichnet wurden. Die Äußerungen enthalten touristische Fragen in englischer Sprache über Straßen, Hotels und Sehenswürdigkeiten der Stadt Karlsruhe [FSS+03]. Insgesamt wurden 9 Sprecher aufgenommen (4 Frauen und 5 Männer).
Die akustischen Modelle wurden beide mit den gleichen Trainingsdaten und äußerungsbasierender Mittelwertsnormalisierung[8] (CMN) und Varianznormalisierung[9] (CVN) trainiert. Die erste Akustik (in Folgenden MS genannt) enthält keine weiteren Optimierungen. Die zweite Akustik (im Folgenden OFS genannt) wurde mit Hilfe von *semi-tied* Kovarianzen (STC) optimiert [Gal99]. Durch eine Transformation des Merkmalsraums werden hier die Achsen so gedreht, dass die Gaußverteilungen mit diagonalisierten Kovarianzen die best mögliche Approximation für die ursprünglichen Gaußverteilungen ergeben, um den Verlust durch die Diagonalisierung der Kovarianzmatrizen zu verringern.
Beide akustischen Modelle wurden zusammen mit einer kontextfreien Grammatik mit 198 Regeln (1408 Knoten mit 1851 Übergängen) verwendet. Das Vokabular hatte eine Größe von 2023 Wörtern.

[7] *engl.* ***M**el **F**requency **C**epstral **C**oefficients*, MFCC, siehe [YW00]
[8] *engl.* ***C**epstral **M**ean **N**ormalization*, CMN
[9] *engl.* ***C**epstral **V**ariance **N**ormalization*, CVN

6.1 Vorverarbeitung

Für die Vorverarbeitung wurde eine spezielle Methode (`FeatureSet->doCE`) implementiert, in der alle Verarbeitungsschritte durchgeführt werden. Dadurch wurde der zusätzliche Aufwand der Skriptverarbeitung reduziert, der allerdings nur einen sehr kleinen Anteil der Rechenzeit beansprucht. Bei der Auswahl der Optimierungen wurde versucht, einen guten Kompromiss zu finden zwischen Geschwindigkeit und Wortakkuratheit. Diese Methode wurde für alle weiteren Evaluierungen verwendet und enthält folgende Optimierungen:

Funktion	FFT	VTLN	Log	DCT	LDA	NOPT	Gesamt
Baseline [ms]	642	252	116	69	1108	103	2290
Optimierung [ms]	39	72	15	11	89	98	324
Beschl.-Faktor	16,5	3,5	7,7	6,3	12,4	1,1	7,1

Tabelle 12: Beschleunigung der Vorverarbeitung

Akustik	MS	OFS
WA Baseline	73,36%	73,77%
WA Optimierung	74,50%	74,77%

Tabelle 13: Einfluß der Optimierung der Vorverarbeitung auf die Wortakkuratheit

Fast-Fourier-Transfromation (FFT)
Die FFT aus der IPP Bibliothek ist sehr ungenau und vorsichtig einzusetzen, da offenbar sehr viele Ganzzahl-Überläufe auftreten. Durch eine vorangehende Normalisierung des Signals auf 2% kann der Fehler allerdings minimiert werden. Weil die FFT einen großen Anteil der Rechenzeit für die Vorverarbeitung besitzt, kann auf dem PDA nicht auf eine Beschleunigung verzichtet werden, sofern eine Echtzeit Erkennung angestrebt wird. Die Berechnung des Hamming-Fensters wurde zusätzlich mit Ganzzahlarithmetik beschleunigt.

Vokal-Track-Längen-Normalisierung (VTLN)
Die optimierte VTLN hat keine Parameter, um die Geschwindigkeit oder Genauigkeit zu beeinflussen. Da die Abweichungen in der Berechnung im Vergleich zur Baseline vernachlässigbar sind, kann sie immer eingesetzt werden.

Approximation des Logarithmus (Log):
Für die Approximation des Logarithmus wurde das verbesserte Polynom gewählt, da es mit Abstand die höchste Genauigkeit liefert und sich das auch

auf die Wortakkuratheit auswirkt. Die lineare Approximation wäre zwar um etwa 75% schneller, aber angesichts des relativ kleinen Anteils an der gesamten Vorverarbeitung ist hier die genauere Variante vorzuziehen. Die Optimierung wurde verwendet, um die Koeffizienten des Spektrums in eine logarithmische Skala zu transformieren.

Matrixmultiplikation (DCT, LDA):
Es wurde sowohl die DCT zur Überführung des Mel-Spektrums in das Mel-Cepstrum als auch die LDA mit der Matrixmultiplikation mit Ganzzahlen beschleunigt. Die Skalierungsfaktoren wurden fest angegeben. Das ist die schnellste Variante, da die Bestimmung der Skalierungsfaktoren komplett wegfällt. Werden die Faktoren allerdings zu hoch vorgegeben, muss mit Überläufen gerechnet werden. Es hat sich allerdings gezeigt, dass der Spielraum während der Evaluierung noch sehr groß war, obwohl sich die fest angegebenen Skalierungsfaktoren an Beobachtungen der verbesserten Abschätzung (int_es) orientierten. Eine negative Beeinflussung der Wortakkuratheit konnte nicht festgestellt werden. Die Beschleunigung der DCT fällt deutlich niedriger aus als die der LDA, da die beteiligten Matrizen kleiner sind und der konstante zusätzliche Aufwand erhalten bleibt, wie beispielsweise das Anlegen von temporären Matrizen.

Restliche Verarbeitungsschritte (NOPT)
Hier sind alle Verarbeitungsschritte zusammengefasst, die nicht direkt optimiert wurden. Die leichte Beschleunigung ist auf die vereinfachte Skriptverarbeitung zurückzuführen.

Tabelle 12 zeigt die Ausgangsbasis im Vergleich zu den ausgewählten Optimierungen. Die Wortakkuratheit wird durch die optimierte Vorverarbeitung nicht beeinträchtigt. Bei der Evaluierung konnte im Vergleich zur Ausgangsbasis sogar bei beiden Akustiken eine Verbesserung um etwa 1% absolut ermittelt werden, wie aus Tabelle 13 entnommen werden kann. Die Verbesserung ist hauptsächlich auf die Normalisierung vor der FFT durch die IPP des Audio-Signals zurückzuführen. Ohne die Normalisierung ist die Wortakkuratheit deutlich schlechter. Allerdings ist dann auch die Genauigkeit der FFT schlechter, wie es in Kapitel 3.2 beschrieben ist.

6.2 Decodierung

	Distanzen	Strahlsuche	Gesamt	WA
Baseline	42.704ms	7.290ms	49.994ms	73,36%
EFVR	26.172ms	4.272ms	30.443ms	73,45%
BBI	11.647ms	6.652ms	18.299ms	72,18%
Int	4.191ms	7.382ms	11.573ms	72,95%
EFVR, BBI	7.215ms	4.044ms	11.260ms	72,14%
BBI, Int	2.585ms	7.106ms	9.691ms	71,45%
EFVR, Int	2.617ms	4.534ms	7.151ms	73,64%
EFVR, BBI, Int	1.574ms	4.210ms	5.784ms	72,00%
EFVR, BBI, Int_s	1.338ms	4.201ms	5.739ms	72,00%
VV, EFVR, BBI, Int	1.569ms	4.312ms	5.880ms	73,32%

Tabelle 14: Beschleunigung der Decodierung mit der MS-Akustik

	Distanzen	Strahlsuche	Gesamt	WA
Baseline	43.857ms	6.582ms	50.439ms	73,77%
EFVR	26.600ms	4.019ms	30.620ms	73,82%
BBI	12.119ms	6.091ms	18.210ms	73,64%
Int	4.201ms	7.510ms	11.710ms	74,77%
EFVR, BBI	7.518ms	3.731ms	11.249ms	73,27%
BBI, Int	2.516ms	7.009ms	9.525ms	73,55%
EFVR, Int	2.554ms	4.246ms	6.800ms	73,95%
EFVR, BBI, Int	1.557ms	4.085ms	5.643ms	72,95%
EFVR, BBI, Int_s	1.337ms	4.190ms	5.527ms	72,95%
VV, EFVR, BBI, Int	1.536ms	4.223ms	5.759ms	72,95%

Tabelle 15: Beschleunigung der Decodierung mit der OFS-Akustik

Bei der Evaluierung wurde die Decodierung in zwei Bereiche aufgeteilt, die Berechnung der Mahalanobis-Distanzen und die Strahlsuche, wobei hier auch alle restlichen Berechnungen hinzugezählt werden. Für die Beschleunigung der Berechnung der Mahalanobis-Distanzen wurde die Ganzzahlarithmetik (Int) verwendet. Als Skalierungsfaktoren wurden bei der MS-Akustik für die Mittelwerte $s_m = 128$ und für die Kovarianzen $s_c = 64$ gewählt, da hier der Ganzzahlenraum gut ausgeschöpft wird. Die Kovarianzen werden dadurch allerdings zu etwa 10% beschnitten. Für die OFS-Akustik wurde $s_c = 16$ gewählt, da hier der Wertebereich der Kovarianzen deutlich höher liegt. Auch hier werden etwas weniger als 10% der Kovarianzen beschnitten. Auf verbesserte Genauigkeit durch Bit-Schieben wurde verzichtet. Die Verbesserung Int_s mit serialisierten Speicherzugriffen wurde nur abschließend in Kombi-

nation mit allen anderen Optimierungen der Decodierung eingesetzt, da hier starke Schwankungen in der Zeitmessung beobachtet wurden. Die Experimente mit Int_s wurden mehrmals durchgeführt, um einen Mittelwert der gemessenen Zeiten zu bilden. Der Vergleich mit Int sollte daher mit Skepsis betrachtet werden. Tendenziell ist aber eine leichte Beschleunigung der Berechnung der Mahalanobis-Distanzen zu beobachten. Der Ablauf der Decodierung ist bis auf die benötigte Zeit identisch mit der Variante Int.
Als weitere Optimierung wurde der BBI Tree (BBI) verwendet, um die Anzahl der auszuwertenden Gaußverteilungen zu reduzieren.
Auch die frühe Reduzierung der Merkmalsvektoren (EFVR) wurde verwendet, um sowohl die Anzahl der auszuwertenden Gaußverteilungen als auch den Aufwand der Strahlsuche zu reduzieren.
In den Tabellen 14 und 15 sind jeweils für die beiden Akustiken die Optimierungen der Decodierung in den verschiedenen Kombinationen festgehalten. Ein großer Unterschied besteht zwischen den Akustiken allerdings nicht, sowohl in der Beschleunigung als auch in dem Verhalten der Wortakkuratheit. Bei den meisten Experimenten wurde die Vorverarbeitung nicht optimiert, um deren Einfluß zu vermeiden. Nur im letzten Experiment wird ein Vergleich mit optimierter Vorverarbeitung gegeben (VV). Die Wortakkuratheit variiert um etwas mehr als 1% absolut um 73,00%. Grundsätzlich konnte über alle Experimente hinweg beobachtet werden, dass der BBI Tree die Wortakkuratheit leicht vermindert. Die EFVR und die Optimierung der Vorverarbeitung bewirken eine leichte Verbesserung. Die Berechnung der Mahalanobis-Distanzen mit Ganzzahlarithmetik verhält sich wechselhaft und kann nicht vorhergesagt werden. Angesichts der Beschleunigung um etwa den Faktor 9 und der nur leicht veränderten Wortakkuratheit des Endergebnisses sollten alle Optimierungen auf dem PDA eingesetzt werden. Die OFS-Akustik besitzt im Durchschnitt aller Optimierungen eine etwas höhere Wortakkuratheit und sollte daher vorgezogen werden.

6.3 Trimmen auf Echtzeit

Mit den oben vorgestellten Optimierungen konnte mit beiden Akustiken nur ein Echtzeitfaktor von ungefähr 6,0 erreicht werden. Die meiste Rechenzeit wird für die Strahlsuche benötigt. Daher wurde mit beiden Akustiken eine weitere Testreihe durchgeführt, bei der die Strahlbreite schrittweise verkleinert wurde, um am Ende einen Echtzeitfaktor von ≤ 1.0 zu erzwingen. Die Angabe des Suchstrahls ist als Faktor zu sehen, der zur Skalierung der JANUS-spezifischen Angaben für den *Master Beam*, *transN* und *morphN* dient. Die Ausgangsbasis und die optimierte Erkennung hatten zunächst einen Faktor von 1,3. In den Tabellen 16 und 17 ist der Verlauf festgehalten und in den Abbildungen 7 und 8 grafisch veranschaulicht, wobei hier von der bereits optimierten Erkennung mit Strahlbreite 1,3 ausgegangen wird. Die Ausgangsbasis ist nicht eingezeichnet, da sie die gleiche Strahlbreite besitzt

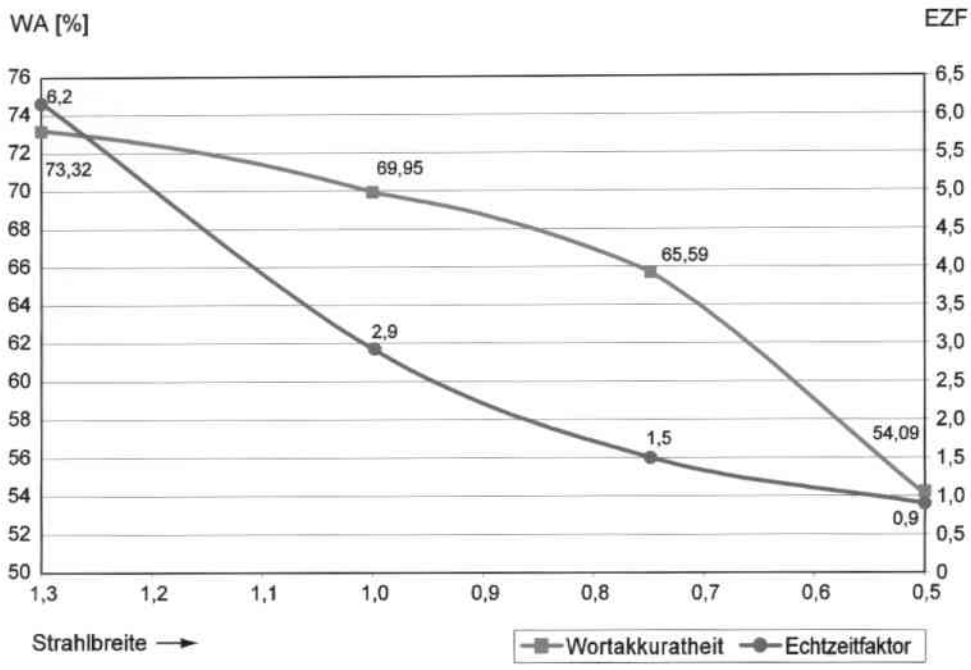

Abbildung 7: Vergleich des Echtzeitfaktors und der Wortakkuratheit in Abhängigkeit der Strahlbreite auf der MS-Akustik

und für den Effekt der Verengung des Suchstrahls nicht relevant ist. Durch die Verengung lässt sich in etwa eine Halbierung des Echtzeitfaktors auf 2,9 erzielen. Wird der Strahl weiter eingeschränkt, sinkt die Wortakkuratheit deutlich ab. Der Strahl sollte also nicht enger als 1,0 gewählt werden. Die MS-Akustik verschlechtert sich durch den engeren Suchstrahl etwas stärker als die OFS-Akustik.

	Strahl	VV	Distanzen	Strahlsuche	EZF	WA
Baseline	1,30	2.290ms	42.704ms	7.290ms	52,3	73,36%
Optimierung	1,30	329ms	1.569ms	4.312ms	6,2	73,32%
Optimierung	1,00	329ms	789ms	1.764ms	2,9	69,95%
Optimierung	0,75	329ms	404ms	718ms	1,5	65,59%
Optimierung	0,50	329ms	227ms	388ms	0,9	54,09%

Tabelle 16: Trimmen der Erkennung auf Echtzeit basierend auf der MS-Akustik

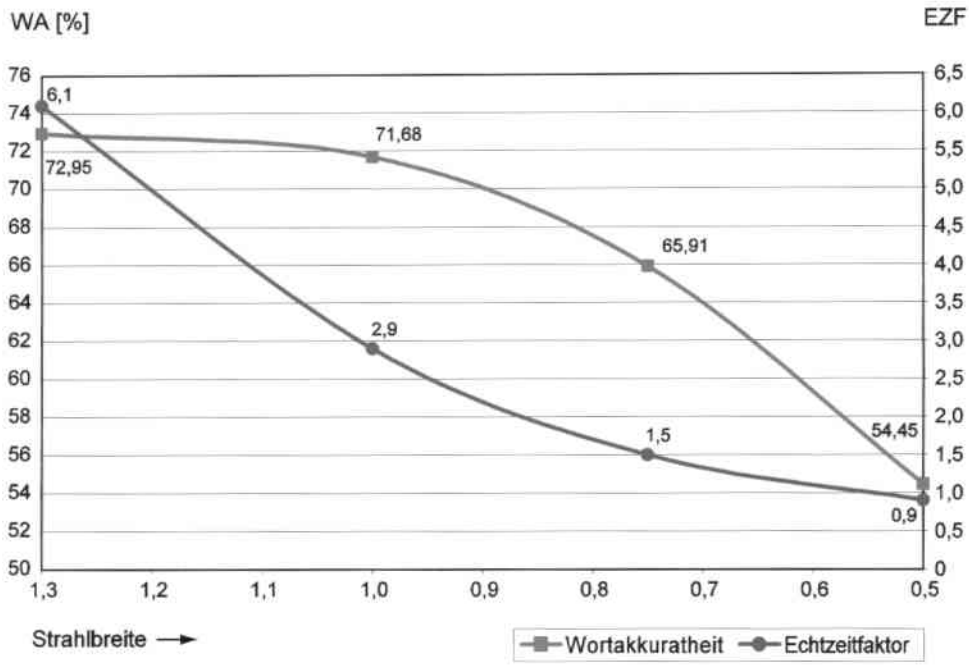

Abbildung 8: Vergleich des Echtzeitfaktors und der Wortakkuratheit in Abhängigkeit der Strahlbreite auf der OFS-Akustik

	Strahl	VV	Distanzen	Strahlsuche	EZF	WA
Baseline	1,30	2.290ms	43.857ms	6.582ms	52,7	73,77%
Optimierung	1,30	329ms	1.536ms	4.223ms	6,1	72,95%
Optimierung	1,00	329ms	887ms	1.716ms	2,9	71,68%
Optimierung	0,75	329ms	456ms	708ms	1,5	65,91%
Optimierung	0,50	329ms	202ms	336ms	0,9	54,45%

Tabelle 17: Trimmen der Erkennung auf Echtzeit basierend auf der OFS-Akustik

7 Zusammenfassung / Ausblick

Der Spracherkenner JANUS konnte erfolgreich auf die Plattform Windows CE 3.0 portiert werden. Mit einem Echtzeitfaktor von deutlich über 50 als Ausgangsbasis war der Erkenner allerdings für echte Anwendungen zu langsam. JANUS wurde mit den hier vorgestellten Optimierungen auf einen Echtzeitfaktor von ungefähr 6,0 beschleunigt, ohne einen signifikanten Verlust an Wortakkuratheit. Bei dem Einsatz der frühen Reduzierung der Merkmalsvektoren (EFVR) konnte in den meisten Fällen sogar eine leichte Verbesserung beobachtet werden.
Durch eine Verkleinerung des Suchstrahls konnte der Echtzeitfaktor noch einmal um mehr als die Hälfte auf 2,9 reduziert werden. Der Verlust an Wortakkuratheit lag bei der MS-Akustik unter 5% relativ zur Ausgangsbasis, bei der OFS-Akustik unter 3%. Um den Spracherkenner mit Hilfe der Strahlbreite auf tatsächliche Echtzeit zu trimmen, mussten allerdings große Einbrüche der Wortakkuratheit von über 25% relativ hingenommen werden. Als größtes Hindernis auf dem Weg zur Echtzeit hat sich dabei die Strahlsuche herausgestellt, die in dieser Arbeit nur indirekt durch die Reduzierung der Merkmalsvektoren um den Faktor 1,6 (OFS-Akustik) bis 1,7 (MS-Akustik) beschleunigt wurde. Hier wäre eine weitere Optimierung anzusetzen, möchte man den Echtzeitfaktor deutlich verringern, ohne dabei die Akustik, das Vokabular oder die Grammatik zu verkleinern.
Die Vorverarbeitung konnte insgesamt etwa um den Faktor 7 beschleunigt werden. Eine weitere Beschleunigung könnte erreicht werden, indem die Vorverarbeitung komplett auf Ganzzahlen umgestellt wird. Dadurch könnte die Umwandlung von Fließkommazahlen in Ganzzahlen und umgekehrt zwischen den Verarbeitungsschritten eingespart werden. Allerdings macht diese Umwandlung nur einen kleinen Teil aus, und die modulare Eigenschaft der Optimierungen ginge verloren. Eine grobe Schätzung des Beschleunigungspotentials läge bei etwa 10% bis 25%, je nach Optimierung.
Den größten Anteil an der Vorverarbeitung haben die FFT und die Matrixmultiplikation, die beide bereits eine effiziente Implementierung besitzen. Eine weitere signifikante Beschleunigung kann hier wohl nicht mehr erreicht werden ohne methodische Veränderung der Vorverarbeitung. Der Flaschenhals ist der Datendurchsatz des Arbeitsspeichers.
Die Berechnung der Mahalanobis-Distanzen wurde um einen Faktor von knapp 30 beschleunigt, davon etwa um den Faktor 10 alleine durch die Verwendung von Ganzzahlarithmetik. Auch hier kommt der PDA an die Leistungsgrenze des Arbeitsspeichers. Das kann daran beobachtet werden, dass zusätzliche Rechenoperationen wie das Bit-Schieben zwischen den Speicherzugriffen keine Geschwindigkeitseinbußen mit sich bringen. Der Prozessor muss also auf den Speicher warten. Darum sind auch die Nachschlagetabellen durch den zusätzlich anfallenden Speicherzugriff langsamer als die tatsächliche Berechnung der Mahalanobis-Distanzen.

Weiterführende Optimierungen müssen daher einen methodischen Ansatz verfolgen, etwa durch eine Verkleinerung der Akustik durch Verklebung von Zuständen der HMMs oder SDCHMMs [JKO04].
Während den Tests bei dieser Arbeit war aufgefallen, dass die Erkennung nicht vollständig unabhängig von der Lautstärke der Eingabe ist. Auch wich die Wortakkuratheit unter den verschiedenen Sprechern der Testdaten sehr stark voneinander ab. Die Wortakkuratheit lag sprecherabhängig zwischen 40% und 90%. Hier könnte möglicherweise durch ein sog. Autoleveling[10] eine bessere Erkennungsleistung erzielt werden. Auch eine vorangehende Audio-Kompression[11] könnte eine Verbesserung bewirken, da verschiedene Sprecher die Konsonanten sehr verschieden laut aussprechen. Auch sollte stärker mit Frequenz-Filtern gearbeitet werden, die unerwünschte Störgeräusche wie Pop-Laute entfernen, die bei plosiven Konsonanten entstehen. Da die Grundfrequenz der Stimme nicht so wichtig ist wie die höher liegenden Formanten, kann die Grenzfrequenz eines Hochpass-Filters sogar darüber angesetzt werden. Möglicherweise ließen sich durch eine intensivere Bearbeitung des Eingangssignals die Unterschiede einzelner Sprecher verringern. Wenn diese Aufbereitung auch für das Training der Akustiken eingesetzt wird, könnte sich dadurch auch die Anzahl der nötigen Codebücher und Gaußverteilungen reduzieren. Dadurch könnte auch eine Speicher- und Geschwindigkeits-Optimierung erreicht werden.

[10] automatische Regulierung der Lautstärke aufgrund des Pegels des Eingangssignals
[11] Reduzierung der Dynamik durch automatische Lautstärkenanpassung im Kurzzeitbereich

8 Literaturverzeichnis

Literatur

[CSMR04] A. Chan, J. Sherwani, R. Mosur, and A. Rudnicky. Four-layer categorization scheme of fast GMM computation techniques in large vocabulary continuous speech recognition systems. In *Proc. of ICSLP 2004*, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213, USA, 2004.

[FGH+97] M. Finke, P. Geutner, H. Hild, T. Kemp, K. Ries, and M. Westphal. The Karlsruhe-Verbmobil speech recognition engine. In *Proc. of ICASSP 1997*, volume 1, pages 83–86, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, Germany; CMU, USA, 1997.

[FSS+03] C. Fügen, S. Stüker, H. Soltau, F. Metze, and T. Schultz. Efficient handling of multiligual language models. In *Proc. of ASRU 2003*, Interactive Systems Labs, University of Karlsruhe, Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2003.

[Gal99] M.J.F. Gales. Semi-tied covariance matrices for hidden markov models. In *IEEE Transactions on Speech and Audio Processing*, volume 7, pages pp. 272–281, 1999.

[JHJK04] S. Jeong, I. Han, E. Jon, and J. Kim. Memory and computation reduction for embedded ASR systems. In *Proc. of ICSLP 2004*, Human Computer Interaction Lab., Samsung Advanced Institute of Technology, 2004.

[JKO04] Gue Jun Jung, Su-Hyun Kim, and Yung-Hwan Oh. An efficient codebook design in SDCHMM for mobile communication environments. In *Proc. of ICSLP 2004*, Division of Computer Science, Department of Electrical Engineering and Computer Science, KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea, 2004.

[Jän02] Klaus Jänich. *Mathematik 2.* Springer, Berlin, ISBN 3-5404-2839-9, 2002.

[MJF+04] F. Metze, Q. Jin, C. Fügen, K. Laskowski, Y. Pan, and T. Schultz. Issues in meeting transcription - the ISL meeting transcription system. In *Proc. of ICSLP 2004*, Interactive Systems Labs, Universität Karlsruhe (TH), Carnegie Mellon University, 2004.

[Nov04] Miroslav Novak. Towards large vocabulary ASR on embedded platforms. In *Proc. of ICSLP 2004*, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598, USA, 2004.

[PK04] Junho Park and Hanseok Ko. Compact acoustic model for embedded implementation. In *Proc. of ICSLP 2004*, ISPL, Department of Electronics Engineering, Korea University, Seoul, Korea, 2004.

[SMFW01] H. Soltau, F. Metze, C. Fügen, and A. Waibel. A one pass-decoder based on polymorphic linguistic context assignment. In *Proc. of ASRU 2001*, Karlsruhe, Germany, 2001.

[ST95] Ernst Günter Schukat-Talamazzini. *Automatische Spracherkennung*. Vieweg Verlag, Braunschweig/Wiesbaden, 1995.

[Tia04] Jilei Tian. Efficient compression method for pronunciation dictionaries. In *Proc. of ICSLP 2004*, Audio-Visual Systems Laboratory, Nokia Research Center, Tampere, Finland, 2004.

[Vas00] Marcel Vasilache. Speech recognition using HMMs with quantized parameters. In *Proc. of ICSLP 2000*, volume 1, pages 441–444, Speech and Audio Systems Laboratory, Nokia Research Center, Tampere, Finland, 2000.

[Vii01] Olli Viikki. ASR in portable wireless devices. In *Proc. of ASRU 2001*, Nokia Research Center, Speech and Audio Systems Laboratory, Tampere, Finland, 2001.

[VISV04] Marcel Vasilache, Juha Iso-Sipilä, and Olli Viikki. On a practical design of a low complexity speech recognition engine. In *Proc. of ICASSP 2004*, pages 113–116, Audio-Visual Systems Laboratory, Nokia Research Center, Tampere, Finland, 2004.

[WBB+03] A. Waibel, A. Badran, A. W. Black, R. Frederking, D. Gates, A. Lavie, L. Levin, K. Lenzo, L. Mayfield Tomokiyo, J. Reichert, T. Schultz, D. Wallace, M. Woszczyna, and J. Zhang. Speechalator: Two-way speech-to-speech translation on a consumer pda. In *Proc. of EUROSPEECH 2003*, pages 369–372, Language Technologies Institute, Carnegie Mellon University, Pittsburgh, PA, USA; Cepstral LLC, USA; Multimodal Technologies Inc., USA; Mobile Technologies Inc., USA, 2003.

[Wel97] Brent Welch. *Praktisches Programmieren in Tcl/Tk*. Markt und Technik, ISBN 3-8272-9529-7, 1997.

[Wos98] Monika Woszczyna. *Fast Speaker Independent Large Vocabulary Continuous Speech Recognition.* PhD thesis, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 1998.

[YW00] Hua Yu and Alex Waibel. Streamlining the front end of a speech recognizer. In *Proc. of ICSLP 2000*, 2000.

[ZGS+03] B. Zhou, Y. Gao, J. Sorensen, D. Déchelotte, and M. Picheny. A hand-held speech-to-speech translation system. In *Proc. of ASRU 2003*, volume 1, pages 664–669, IBM T.J. Watson Research Center, Yorktown Heights, New York 10598, 2003.

A Dokumentation der einzelnen Optimierungen

A.1 Matrixmultiplikation mit Ganzzahlarithmetik

Für die optimierte Matrixmultiplikation gibt es mehrere Implementierungen. Sie kann bei der Multiplikation eines Features mit einer Fließkomma-Matrix der Klasse `FMatrix` durch das zusätzliche Argument `-mod` ausgewählt werden.

```
<featureSet> matmul <new_feature> <source_feature> <fmatrix>
[-cut cut] [-mod mod]
```

Berechnet die Matrixmultiplikation `source_feature` $*$ `matrix`, wobei

`featureSet`	Name des Feature-Sets	
`new_feature`	Name des neuen Feature-Objekts	
`source_feature`	Name des Quell-Feature-Objekts	
`fmatrix`	Name eines Fließkommamatrix-Objekts	
`cut`	übernehme die ersten n Spalten	
`mod`	wähle den Algorithmus aus	
	`std`	keine Optimierung (voreingestellt)
	`intn`	einfache Abschätzung mit gemeinsamen Skalierungsfaktor
	`intns`	einfache Abschätzung mit verschiedenen Skalierungsfaktoren
	`inte`	verbesserte Abschätzung mit gemeinsamen Skalierungsfaktor
	`intes`	verbesserte Abschätzung mit verschiedenen Skalierungsfaktoren
	`intra`	individuelle Skalierung pro Reihe
	`intera`	individuelle Skalierung pro Reihe mit verbesserter Abschätzung

A.2 FFT mit Intel Performance Primitives

Die FFT zur Berechnung des Leistungsspektrums eines Signals wird mit Hilfe der Intel Performance Primitives analog zur Methode `spectrum` der Klasse `FeatureSet` verwendet.

```
<featureSet> IPPspectrum <new_feature> <source_feature> <win>
[-shift shift] [-normalize max]
```

Berechnet wird das Leistungsspektrum von `source_feature`, wobei

`featureSet`	Name des Feature-Sets

`new_feature`	Name des neuen Feature-Objekts
`source_feature`	Name des Quell-Feature-Objekts
`win`	Größe des Zeitfensters (in ms)
`shift`	Verschiebung des Zeitfensters
`max`	$\in (0, 1]$ und gibt das Normalisierungsniveau an

Bevor die FFT durchgeführt wird, legt diese Methode ein Hamming-Fenster über den Zeitbereich. Die Berechnung des Hamming-Fensters wurde mit Ganzzahlarithmetik beschleunigt und stammt nicht aus den Intel Performance Primitives. Des Weiteren wurde eine optionale Normalisierung des Quell-Feature-Objekts aus Geschwindigkeitsgründen direkt integriert. Für die Berechnung der Normalisierung wird keine Kopie der Daten erstellt, sodass die Änderung permanent auf dem Quell-Feature-Objekt gespeichert wird. Die Normalisierung steht auch für die Standard Methode `spectrum` zur Verfügung.

A.3 Schnelle Approximation der Logarithmus-Funktion

Die schnelle Approximation der Logarithmus-Funktion wurde als Ersatz für die Methode `log` der Klasse `FeatureSet` implementiert. Es stehen vier Versionen zur Verfügung:

Approximation mit verbessetem Polynom:
`<featureSet> fastlog <new_feature> <source_feature> <m> <a>`

Approximation mit Taylorpolynom:
`<featureSet> tfastlog <new_feature> <source_feature> <m> <a>`

Lineare Approximation mit einer Geraden:
`<featureSet> lfastlog <new_feature> <source_feature> <m> <a>`

Grobe Approximation nur durch den Exponenten der Fließkommazahl:
`<featureSet> ufastlog <new_feature> <source_feature> <m> <a>`

Berechnet wird jeweils $approx\{\mathtt{m} \cdot log_{10}(\mathtt{source\_feature} + \mathtt{a})\}$, wobei

`featureSet`	Name des Feature-Sets
`new_feature`	Name des neuen Feature-Objekts
`source_feature`	Name des Quell-Feature-Objekts
`m`	$\in \mathbb{R}$
`a`	$\in \mathbb{R}$

A.4 Optimierung der Vokal-Trakt-Längen-Normalisierung

Die VTLN wurde durch Optimierung des Quellcodes der Standard Methode `VTLN` der Klasse `FeatureSet` beschleunigt. Da einige Multiplikationen durch

mehrfache Additionen ersetzt wurden, weicht das Ergebnis geringfügig ab. Deshalb ist diese Optimierung optional, auch wenn sich im Test gezeigt hat, dass der Unterschied zu vernachlässigen ist.

```
<featureSet> VTLN <new_feature> <source_feature> <ratio>
[-min min] [-max max] [-edge edge] [-mod mod]
```

Führt die VTLN auf `source_feature` durch, wobei

`featureSet`	Name des Feature-Sets
`new_feature`	Name des neuen Feature-Objekts
`source_feature`	Name des Quell-Feature-Objekts
`ratio`	Warp-Faktor
`min`	Minimaler Warp-Faktor
`max`	Maximaler Warp-Faktor
`edge`	Eckpunkt für stückweises Warpen
`mod`	Warping-Modus: `lin`, `nonlin`, `optlin`

Der Warping-Modus `optlin` ist die optimierte Version von `lin`.

A.5 Frühe Reduzierung der Merkmalsvektoren

Die frühe Reduzierung der Merkmalsvektoren (EFVR) wurde als neue und eigenständige Methode der Klasse `FeatureSet` implementiert.

```
<featureSet> EFVR <new_feature> <source_feature> <threshold>
[-weight weight_feature>] [-boost boost]
```

Führt die EFVR auf `source_feature` durch, wobei

`featureSet`	Name des Feature-Sets
`new_feature`	Name des neuen Feature-Objekts
`source_feature`	Name des Quell-Feature-Objekts
`threshold`	Schwellwert, ab dem zwei aufeinander folgende Merkmalsvektoren für ähnlich befunden werden
`weight_feature`	Name des Feature-Objekts, in welches die Gewichte der neuen Merkmalsvektoren gespeichert werden
`boost`	Verstärkungswert für die Gewichte der Vektoren (voreingestellt 1,0)

Der Schwellwert hängt von der Größe der Zeitfenster und deren Verschiebung pro Merkmalsvektor ab. Er muss empirisch bestimmt werden und besitzt einen Wertebereich von 0 (=keine Vektoren werden reduziert) bis etwa 1,0. Bei größeren Werten als 1,0 werden zu viele Vektoren zusammengefasst, die nicht zu dem selben Phonem gehören. Bei einer Größe von 20ms für ein Zeitfenster und einer Verschiebung von 10ms bewegt sich ein sinnvoller

Schwellwert zwischen 0,25 und 0,75.
Das Objekt `weight_feature` wird automatisch angelegt und muss dem Decoder in der From `spass run -weight weight_feeature` übergeben werden.

A.6 Bewertungsfunktion mit Ganzzahlarithmetik und Nachschlagetabellen

Es wurden mehrere Verfahren zur Beschleunigung der Bewertungsfunktion implementiert. Dabei kamen zwei verschiedene Methoden zum Einsatz: Ganzzahlarithmetik und Quantisierung in Kombination mit Nachschlagetabellen. Um eine Bewertungsfunktion auszuwählen und zu konfigurieren, wurde die Methode `setScorFct` der Klasse `SenoneSet` um verschiedene Parameter erweitert.

```
<senoneSet> setScoreFct <name> [-scaleMean mScale]
[-scaleCoVar cvScale] [-quantMean mQuant] [-quantCoVar cvQuant]
[-overwrite overwrite]
```

Setzt die Parameter für die Bewertungsfunktion, wobei

`senoneSet`	Name des Senone-Sets
`name`	Name der der Bewertungsfunktion, neu hinzugekommen sind:
	`int` Bewertungsfunktion mit 32 Bit Ganzzahlarithmetik
	`ints` Bewertungsfunktion mit 16/32 Bit Ganzzahlarithmetik und serialisiertem Speicherzugriff
	`quant` Bewertungsfunktion mit temporären Tabellen
	`quant2` Bewertungsfunktion mit globalen Tabellen
`cvScale`	Skalierungsfaktor für die Kovarianz Werte
`mScale`	Skalierungsfaktor für die Mittelwerte und die Merkmalsvektoren
`cvQuant`	Quantisierungsstufen der Kovarianzen für die Tabellen
`mQuant`	Quantisierungsstufen der Mittelwerte für die Tabellen
`overwrite`	$\in [0;1]$, $1 \hat{=}$ das alte Codebuch wird überschieben

Nutzt man die `overwrite` Option, so können keine anderen Bewertungsfunktionen mehr ausgeführt werden, ohne das Codebuch neu zu laden. Dafür wird der Speicher für das Codebuch nicht doppelt belegt. Auf dem PDA unter WinCE ist diese Funktion bei größeren Akustiken (>4MB) notwendig.
Die Grenzen der Wertebereiche für die Quantisierung der Codebücher wurde empirisch ermittelt und beim Setzen von `quant` oder `quant2` automatisch angewandt. Für die Mittelwerte und Merkmalsvektoren gilt $m_i, x_i \in [-6, 6]$ und die Kovarianzen $c_i i \in [0.01, 6]$.

A.7 Komplette Vorverarbeitung in einer Funktion

Die beste Kombination von Optimierungen der Vorverarbeitung, wie sie in dieser Arbeit verwendet wurde, ist in einer einzigen Funktion `doCE` zusammengefasst, um den zusätzlichen Aufwand der Skriptverarbeitung zu minimieren. Nicht enthalten ist die EFVR.

`<featureSet> doCE <source_feature>`

Führt die komplette Vorverarbeitung durch.

`featureSet`	Name des Feature-Sets
`source_feature`	Name des Quell-Features

Die zu evaluierende Audio-Datei muss davor in `source_feature` geladen werden. Anschließend wird das `source_feature` der Funktion `doCE` übergeben. Nach Beendigung dieser Funktion kann direkt zur Decodierung übergegangen werden. Das `source_feature` wird dabei zerstört.

Beispiel:

```
⋮
Initialisierung
⋮
featureSet readADC ADC /path/file.adc
featureSet doCE ADC
spass run
spass.stab trace
```

A.8 Hilfsfunktion Fehlermatrix

Diese Funktion dient dazu, zwei Matrizen, die auf unterschiedlichen Wegen berechnet wurden, zu vergleichen. Sie berechnet die Fehlermatrix, also die Differenz der beiden Matrizen, und gibt eine Statistik über deren Koeffizienten aus.

`<FMatrixA> errdiff <FMatrixB>`

Vergleicht zwei Matrizen **A** und **B** und gibt eine Statistik über die Abweichung aus, wobei

`FMatrixA`	Name eines Objekts vom Typ `FMatrix`
`FMatrixB`	Name eines zweiten Objekts vom Typ `FMatrix`, mit dem verglichen wird

Diese Funktion arbeitet auf zwei Fließkomma-Matrizen (`FMatrix` Objekten) **A** und **B**, die gleiche Dimensionen ($m \times n$) besitzen müssen. Es wird eine dritte, temporäre Matrix **C** erstellt, die mit den Differenzen der einzelnen Koeffizienten gefüllt wird:

$$\mathbf{C} = \mathbf{A} - \mathbf{B}$$

Anschließen wird folgende Statistik ausgegeben:

- Dimensionen und gesamte Anzahl der Koeffizienten zur Kontrolle
- durchschnittliche Abweichung eines Koeffizienten absolut
  $\bar{\delta}_{avg} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}-b_{ij}|}{mn}$
- durchschnittliche Abweichung eines Koeffizienten relativ zu **A**
  $\delta_{avg} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n}\frac{|a_{ij}-b_{ij}|}{a_{ij}}}{mn}$
- maximale Abweichung eines Koeffizienten absolut
  $\bar{\delta}_{max} = max\{|a_{ij} - b_{ij}|\}$
- maximale Abweichung eines Koeffizienten relativ zu **A**
  $\delta_{max} = max\left\{\frac{|a_{ij}-b_{ij}|}{a_{ij}}\right\}$
- Position innerhalb der Matrix mit der maximalen, absoluten Abweichung

Durch diese Angaben kann die Genauigkeit einer Optimierung wie der Matrixmultiplikation mit Ganzzahlen empirisch gemessen und beurteilt werden.

A.9 Hilfsfunktion Matrix-Histogramm

Eine weitere nützliche Funktion ist die Histogramm Funktion für Matrizen:

```
<FMatrixA> histogram [-segments segments]
```

Erstellt ein Histogramm auf mehreren Segmenten, wobei

`FMatrix`	Name eines beliebigen `FMatrix` Objektes
`segments`	Anzahl der Segmente, in die die Matrix aufgeteilt werden soll

Die Funktion sucht das Minimum und das Maximum der Matrix und unterteilt diesen Wertebereich in die angegebene Anzahl von Segmenten. Anschließend wird für jedes Segment die Anzahl der dazugehörigen Koeffizienten aus der Matrix bestimmt und ausgegeben. Dadurch kann die Verteilung der Koeffizienten im Wertebereich beobachtet werden.

A.10 Funktionen für die Evaluierung auf dem PDA

Die Windows CE Version von JANUS unterstützt nicht die Skriptsprache TCL, sondern arbeitet mit einem integrierten Ersatz, der aber nur eine Teilmenge der Funktionalität von TCL enthält. Um eine Evaluierung durchführen zu können, wurden zwei Funktionen implementiert.
Die Angabe von `loop:` an einer beliebigen Stelle im TCL Skript bewirkt, dass das Skript nach der letzten Zeile statt zu beenden wieder an diese Position zurück springt. Diese Funktion sollte nur in Kombination mit `scanADC` verwendet werden, da sonst kein Abbruchkriterium für diese Schleife existiert.
Die Methode `readADC` der Klasse `FeatureSet` zum Einlesen der Audio-Dateien kann durch `scanADC` ersetzt werden. `scanADC` liest die angegebene Text-Datei mit einer Liste von zu evaluierenden Äußerungen ein und vergleicht sie mit der ebenfalls angegebenen Score Datei, die alle bereits evaluierten Äußerungen enthält. Es wird daraufhin die nächste, noch nicht evaluierte Audio-Datei geladen. Wenn alle Audio-Dateien evaluiert wurden, bericht `scanADC` die Schleife mit einer entsprechenden Meldung ab.

```
featureSet scanADC <new_feature> <path> <fileLst> [-skip scoreFile]
```

Liest die nächste zu evaluierende Audio-Datei, wobei

`featureSet`	Name des Feature-Sets
`new_feature`	Name des neuen Feature-Objekts
`path`	absoluter Pfad zu den Audio-Dateien
`fileLst`	Textdatei mit allen zu evaluierenden Äußerungen
`scoreFile`	Datei mit Äußerungen, die übersprungen werden sollen

Normalerweise wird als `scoreFile` die gleiche Datei angegeben, in die die Ergebnisse herausgeschrieben werden. Dafür nutzt man die folgende Funktion:

```
spass.stab evaltrace <scoreFile>
```

Schreibt das Ergebnis der Decodierung in die angegebene Datei, wobei

`spass.stab`	Name der Decoder Instanz
`scoreFile`	Dateiname der Ausgabedatei

Eine solche Schleife sollte wie folgt aufgebaut sein:

```
⋮
Initinalisierung
⋮
loop:
featureSet scanADC ADC /path /path/fileLst.org -skip /path/score.txt
⋮
Vorverarbeitung
⋮
Scoring
⋮
spass.stab evaltrace /path/score.txt
```

Zu beachten ist, dass alle Audio-Dateien in einem Verzeichnis auf dem PDA gespeichert sein müssen. Daher ist die Menge der Äußerungen begrenzt. Auf einem PDA mit 64MB RAM und ohne Erweiterungskarte lassen sich daher ungefähr 20MB Audio-Dateien auf einmal evaluieren. Es ist aber möglich, die Evaluierung zu unterbrechen, die Audio-Dateien auszutauschen und JANUS erneut zu starten. Die Log-Datei sowie die Datei `scoreFile` werden beim nächsten Start nicht gelöscht sondern weitergeführt.