Weiterentwicklung des Neurosimulators FAUN durch Softwarereengineering und Einsatz optimierter BLAS-Bibliotheken

DIPLOMARBEIT

zur Erlangung des Grades eines Diplom-Ökonomen der

Wirtschaftswissenschaftlichen Fakultät der Universität Hannover

vorgelegt von

Matthias Kehlenbeck

[redacted]

Erstprüfer:
Professor Dr. Michael H. Breitner

Hannover, den 16. August 2004

Inhaltsverzeichnis

Kapitel 1

Einleitung

Aus der Perspektive der Wirtschaftsinformatik kann die wissenschaftliche Literatur zum Thema künstliche Intelligenz in symbolistische und konnektionistische Ansätze untergliedert werden. Während Anhänger des Symbolismus die Auffassung vertreten, dass sich Intelligenz auf die Fähigkeit zur Symbolverarbeitung zurückführen lässt,[1] sind Anhänger des Konnektionismus der Meinung, dass sie durch die komplexe Vernetzung von Schaltelementen entsteht. Da Rechner sowohl zur Verarbeitung von Symbolen, als auch zur Simulation von Netzwerken geeignet sind, wird ihnen zugesprochen, mit der entsprechenden Software auch Intelligenz entwickeln zu können.[2] Folglich wird sowohl an Symbolverarbeitungssystemen, als auch an Neurosimulatoren geforscht. Ob Rechner mit diesen Programmen heute oder in Zukunft als intelligent zu bezeichnen sind, ist eine philosophische Frage.[3] Entscheidend ist, dass mit ihrer Hilfe Probleme gelöst werden können. Ihre Weiterentwicklung verbessert somit das wissenschaftliche Instrumentarium.

Im Folgenden werden Ansätze zur Weiterentwicklung des Neurosimulators FAUN beschrieben. Die Leistungsfähigkeit eines Neurosimulators wird maßgeblich durch die Angemessenheit der verwendeten Modelle und die Ausschöpfung der verfügbaren Rechenkapazitäten bestimmt. Die grundsätzliche Funktionalität des Programms soll unberührt bleiben. Es wird also Softwarereengineering betrieben.[4] Damit treten Überlegungen zur Effizienz und nicht zur Effektivität in den Mittelpunkt der Betrachtung. Die Ausführungszeit des Programms soll möglichst stark reduziert werden.

[1] Vgl. Newell und Simon (1976), S. 115–117.
[2] Vgl. Eraßme (2004), S. 48–105.
[3] Vgl. Eraßme (2004), S. 176–316.
[4] Vgl. Stahlknecht und Hasenkamp (2002), S. 324–326.

Um den Einstieg in die Materie zu erleichtern, werden im nächsten Kapitel die Begriffe der Funktionsapproximation und der Neurosimulation erläutert. Weiterhin enthält es eine Beschreibung der unterstützten Netzwerktypen und ihrer Implementation, auf welche an späterer Stelle zurückgegriffen wird. Im dritten Kapitel werden Möglichkeiten zur Optimierung der Ausführungszeit diskutiert. Dazu werden verschiedene Compiler beurteilt und besonders leistungskritische Routinen identifiziert. Die Beurteilung der Compiler erfordert entsprechende Gütekriterien. Sie werden entwickelt und anschließend gemessen. Dabei wird auch der Beispielfundus von FAUN berücksichtigt und durch zwei zusätzliche Aufgaben erweitert. Die Identifikation der leistungskritischen Routinen orientiert sich stark am Quelltext. Es wird untersucht, für welche der verwendeten Operationen sich der Einsatz optimierter Unterprogramme anbietet und für welche davon abzuraten ist. Außerdem wird auf Möglichkeiten hingewiesen, wie die einzelnen Algorithmen besonders effizient für den gegenwärtig eingesetzten Compiler implementiert werden können. Im vierten Kapitel wird überprüft, für welche Operationen überhaupt optimierte Unterprogramme zur Verfügung stehen. Es enthält außerdem einen Überblick über die entsprechenden Bibliotheken. Die Leistungsunterschiede zwischen ihnen werden anhand von Benchmarks veranschaulicht. Dabei werden die Bibliotheken nicht nur isoliert, sondern auch in Verbindung mit den unterschiedlichen Compilern verglichen. Um einen Einblick in die Funktionsweise dieser Bibliotheken zu ermöglichen, wird im fünften Kapitel auf unterschiedliche Ansätze zu deren Implementation eingegangen. Sie stellen ein interessantes Studienobjekt dar. Schließlich enthält das sechste Kapitel eine Zusammenfassung der Ergebnisse sowie einen Ausblick.

Kapitel 6

Schlussbetrachtung

Mit der Verwendung einer besonders effizienten Programmiersprache, dem Gebrauch optimierter Unterprogramme und dem Einsatz eines Hochleistungsoptimierungsverfahrens wurden beim Entwurf des Neurosimulators FAUN ausgezeichnete Vorraussetzungen für Effizienz geschaffen. Zu Beginn des Reengineeringprozesses wurde das Optimierungspotential daher intuitiv als überschaubar eingeschätzt. Um so überraschender war, dass sich die Ausführungszeit bereits durch die Verwendung eines moderneren Compilers deutlich reduzieren ließ. Der gegenwärtig eingesetzte Compiler unterstützt aufgrund seines relativ hohen Alters nicht den vollen Befehlsumfang aktueller Prozessoren. Darüber hinaus hat der entsprechende Optimierer Schwierigkeiten mit dem Quelltext von FAUN. Die aktuellen Compiler erstellen durchweg schnellere Versionen. Allerdings gibt es in Bezug auf deren Effizienz keinen eindeutigen Sieger, da sie abhängig von der gewählten Aufgabe ist. Um die Auswahl eines aktuellen Compilers zu erleichtern, wurden neben der Effizienz auch die diagnostischen Fähigkeiten, die unterstützten Sprachelemente und der Preis verschiedener Produkte gemessen. Hier ist zwischen den Vor- und Nachteilen abzuwägen.
Die Untersuchung der besonders leistungskritischen Routinen hat ergeben, dass es zwar möglich wäre, noch weitere Unterprogramme der BLAS zu ihrer Berechnung einzusetzen, die Effizienz des Programmes dadurch jedoch nicht verbessert, sondern vielmehr verschlechtert würde. Der Grund dafür ist, dass ihre Verwendung ein Aufspalten der Routinen erfordern würde. Dadurch würde jedoch auch unnötiger Datentransfer entstehen. In der Regel ist es aber genau dieser Datentransfer, der die Ausführung einer Routine verzögert.

Anhang F

Übersicht der Testrechner

Tabelle F.1: Übersicht der Testrechner

Name	Prozessor	Taktrate	L1-Cache	L2-Cache	RAM
ATHLON XP-M	AMD Athlon^{TM}XP-M 2800+	2.133 MHz	64/64 KB	512 KB	512 MB, 400 MHz
PENTIUM III	Intel® Pentium® 3	933 MHz	16/16 KB	512 KB	256 MB, 133 MHz
PENTIUM IV	Intel® Pentium® 4	2.666 MHz	8/12 KB	512 KB	1024 MB, 333 MHz

Der L1-Cache ist bei allen Testrechnern in einen Daten- und einen Instruktionencache aufgeteilt.

Alle Angaben sind den Internet-Seiten der Hersteller entnommen.

Anhang G

Rechenzeiten bei Verwendung unterschiedlicher Compiler

Abbildung G.1: Rechenzeiten des BASF Optionen-Beispiels (ATHLON XP-M)

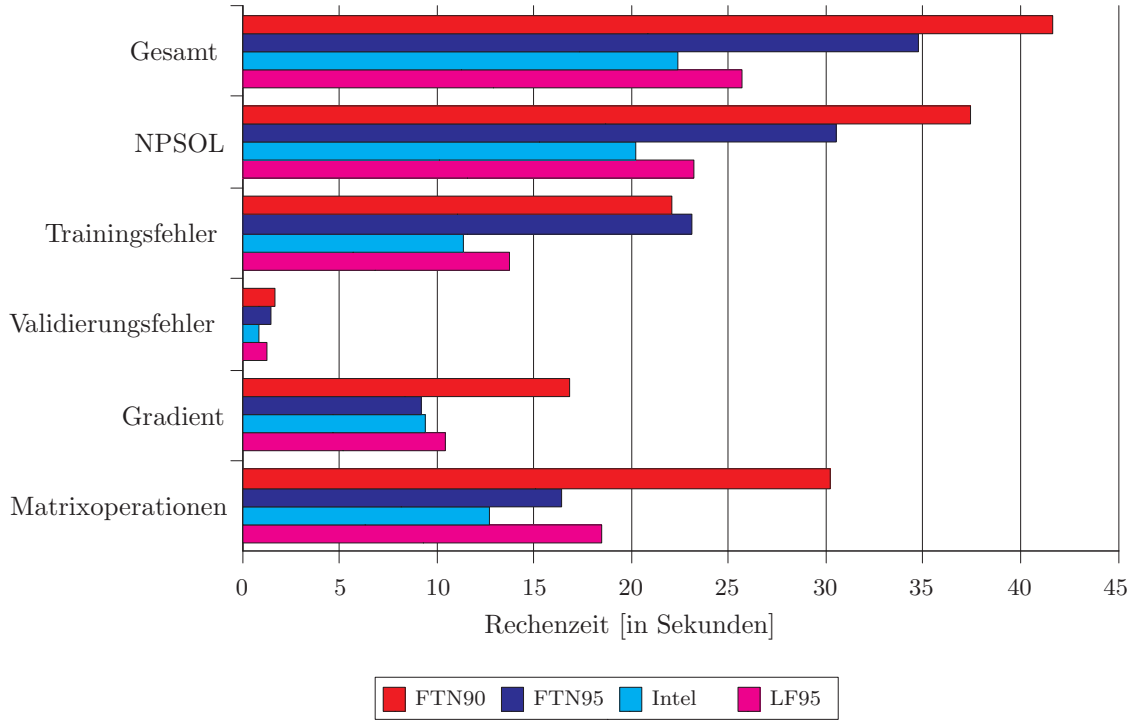


Abbildung G.2: Rechenzeiten des Euro-Dollar Optionen-Beispiels (ATHLON XP-M)

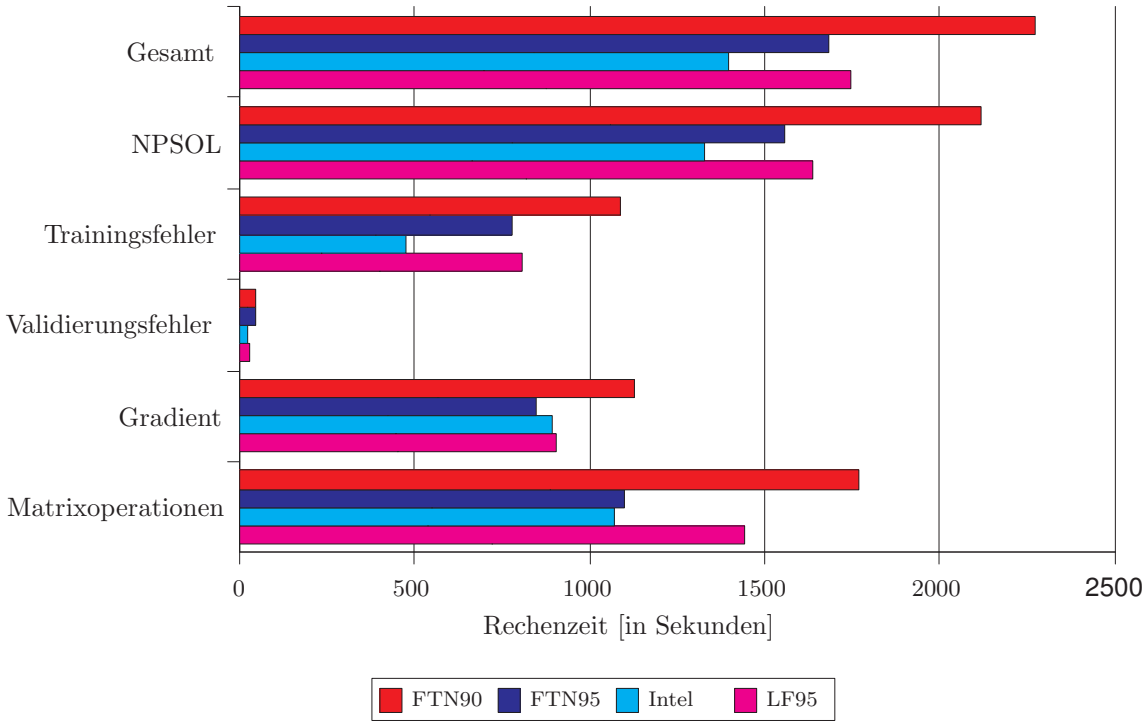


Abbildung G.3: Rechenzeiten des Pilze-Beispiels (ATHLON XP-M)

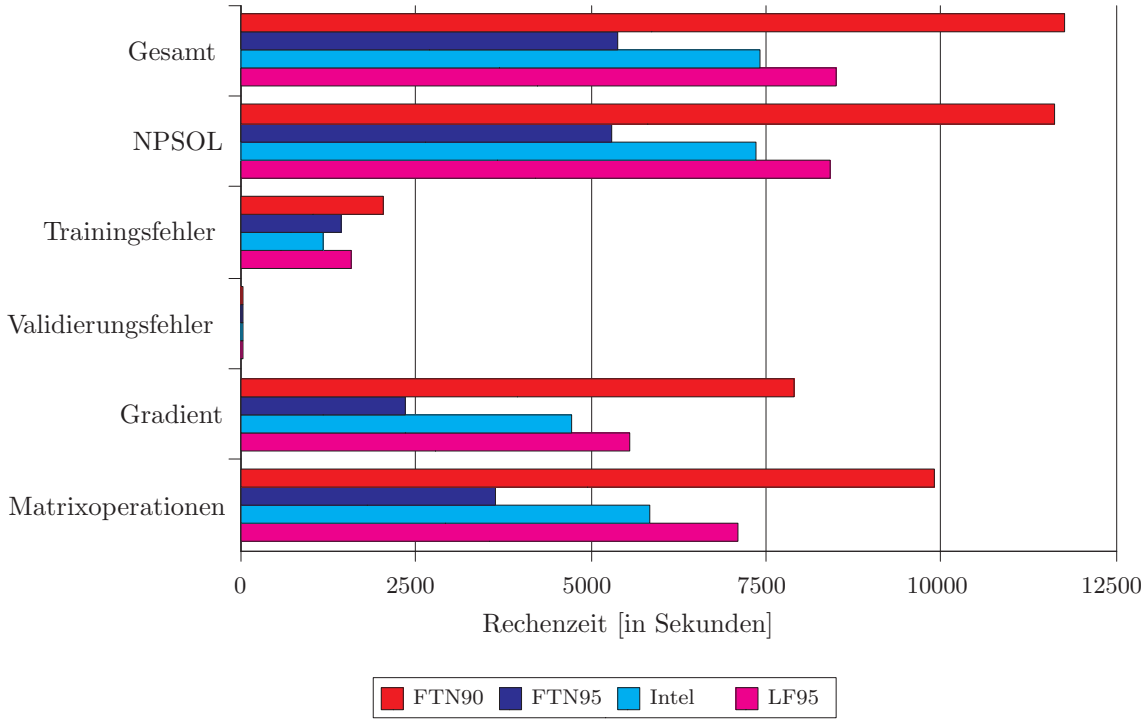


Abbildung G.4: Rechenzeiten des Space Shuttle Guidance-Beispiels (ATHLON XP-M)

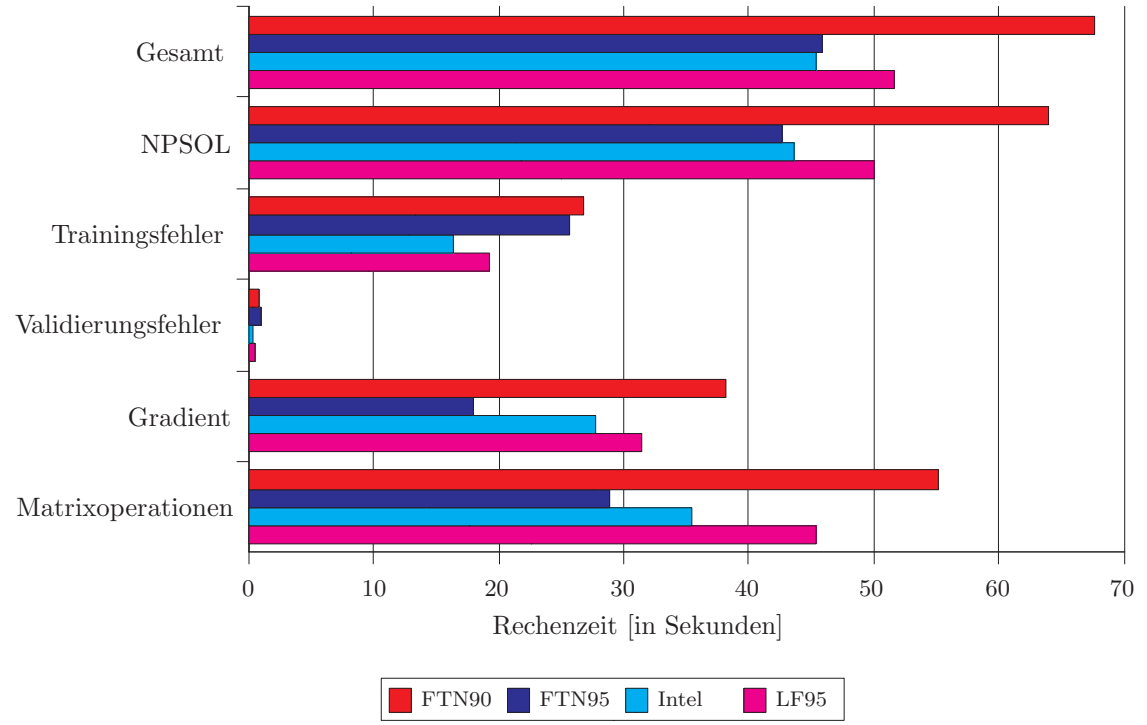


Abbildung G.5: Rechenzeiten des Ziffern-Beispiels (ATHLON XP-M)

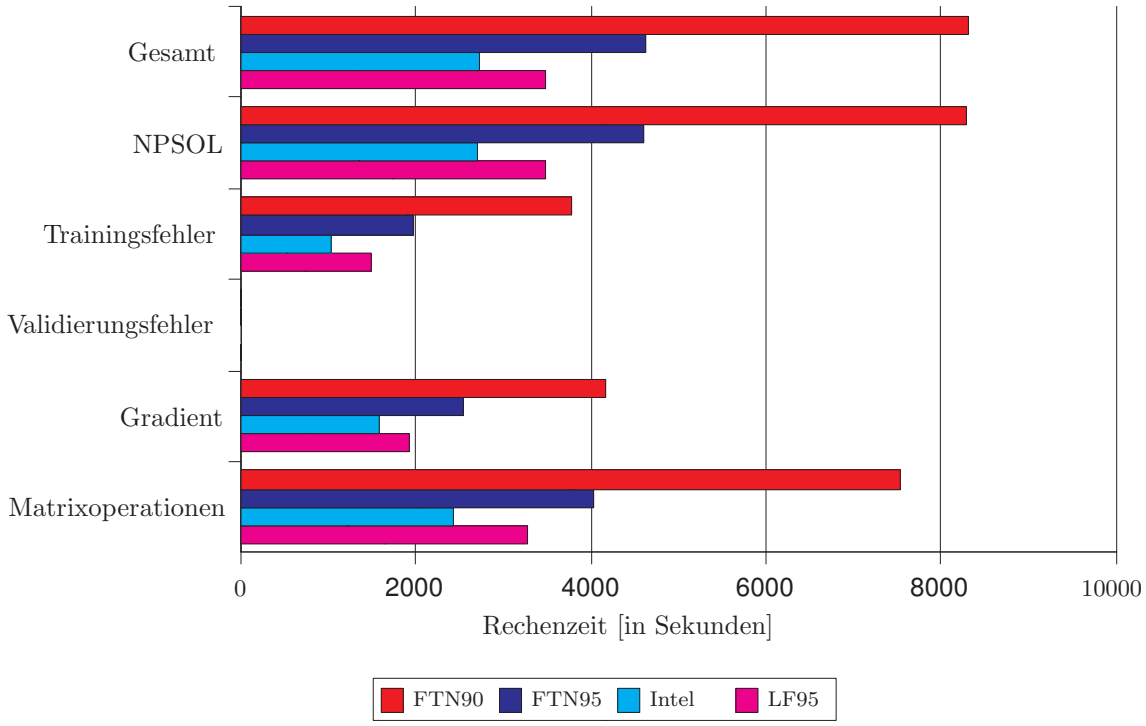


Abbildung G.6: Rechenzeiten des Zinsprognose-Beispiels (ATHLON XP-M)

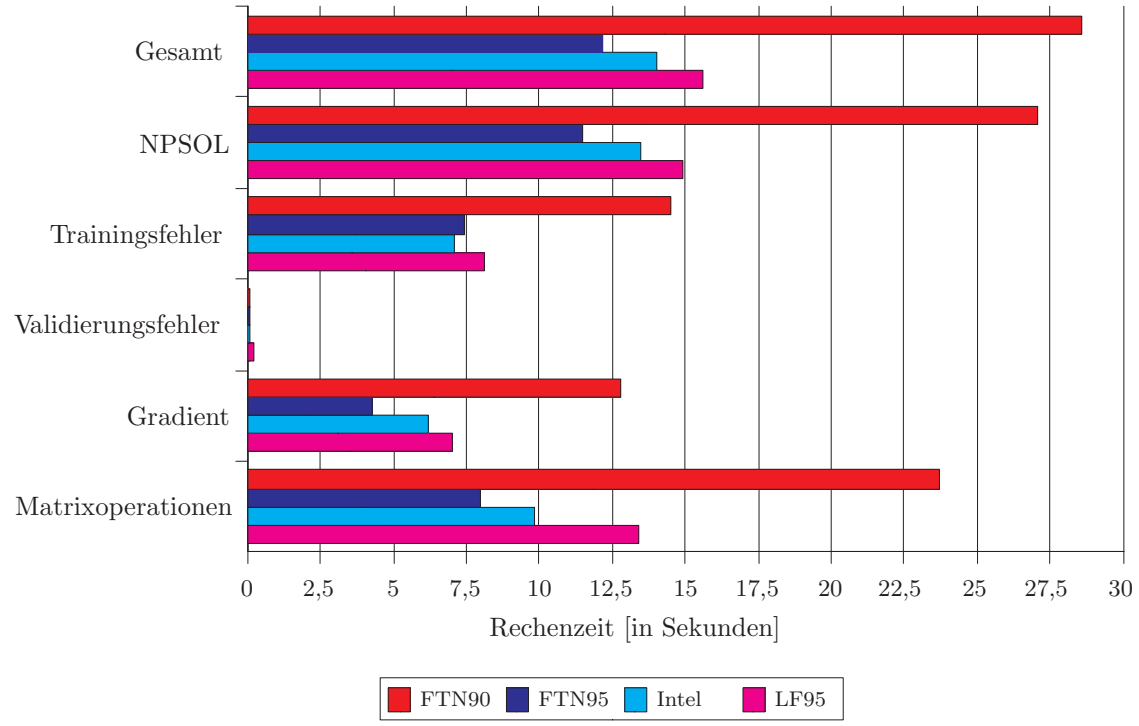


Abbildung G.7: Rechenzeiten des BASF Optionen-Beispiels (PENTIUM III)

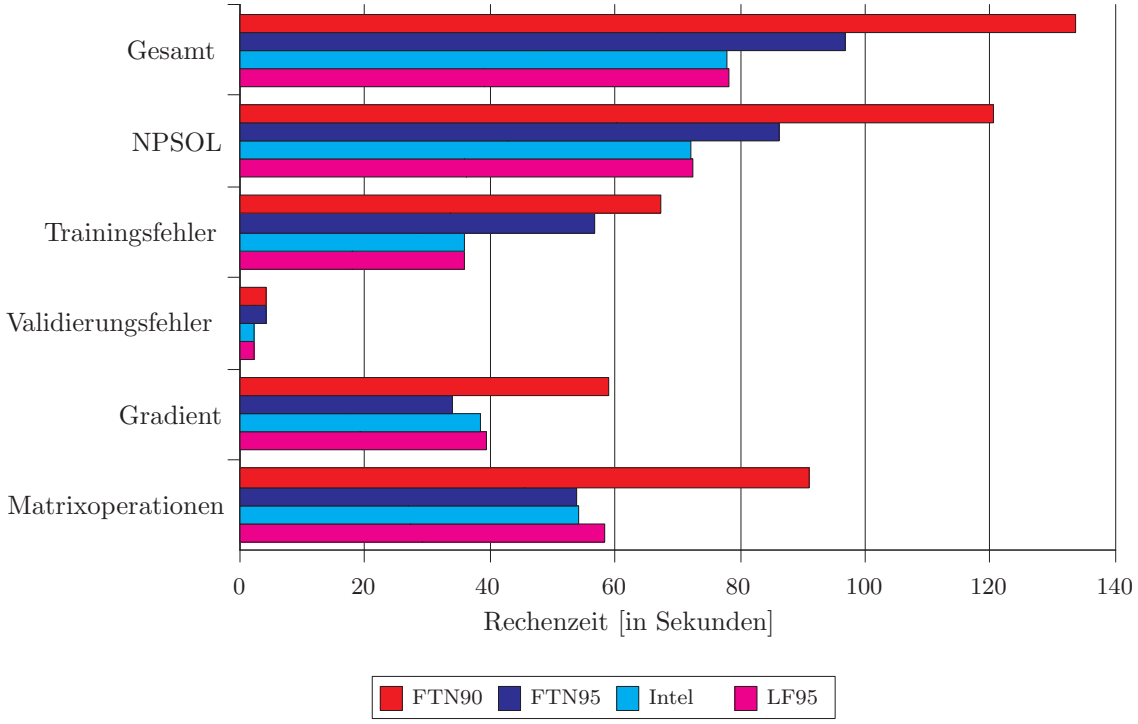


Abbildung G.8: Rechenzeiten des Euro-Dollar Optionen Beispiels (PENTIUM III)

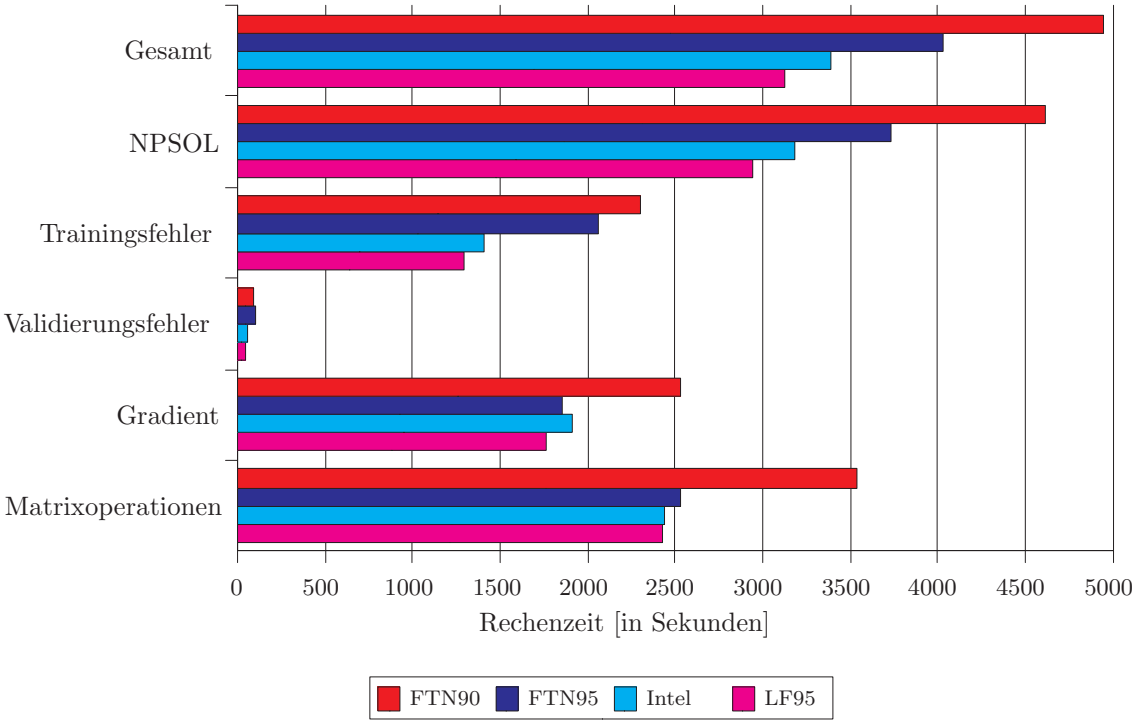


Abbildung G.9: Rechenzeiten des Pilze-Beispiels (PENTIUM III)

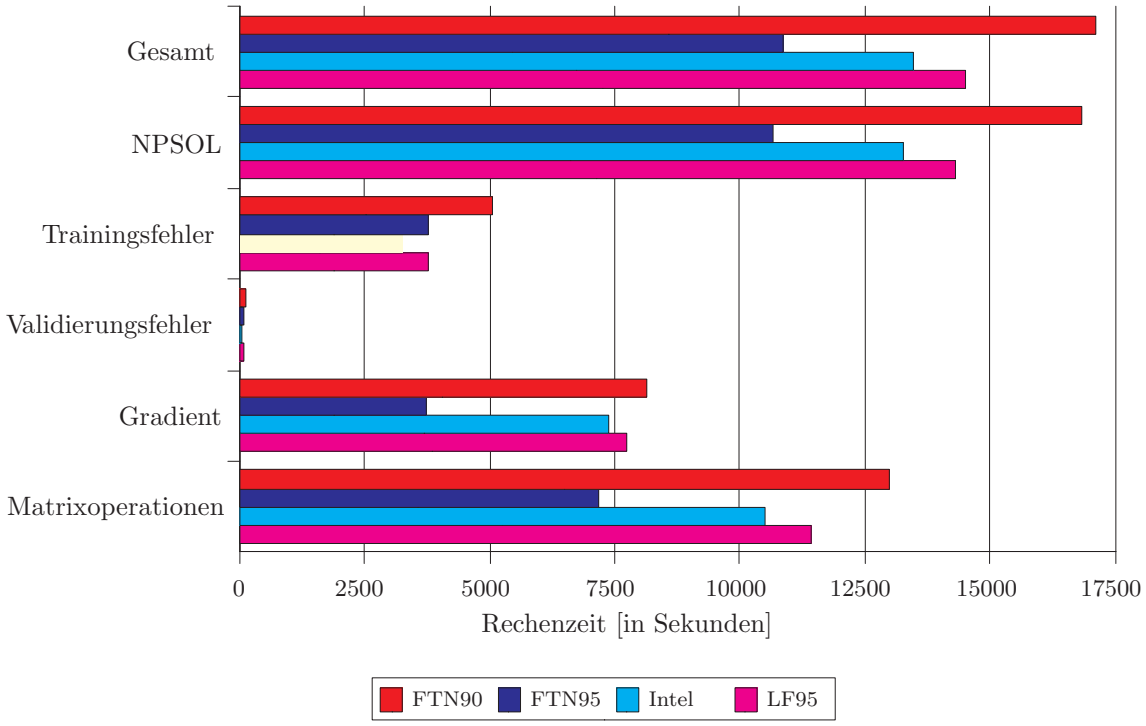


Abbildung G.10: Rechenzeiten des Space Shuttle Guidance-Beispiels (PENTIUM III)

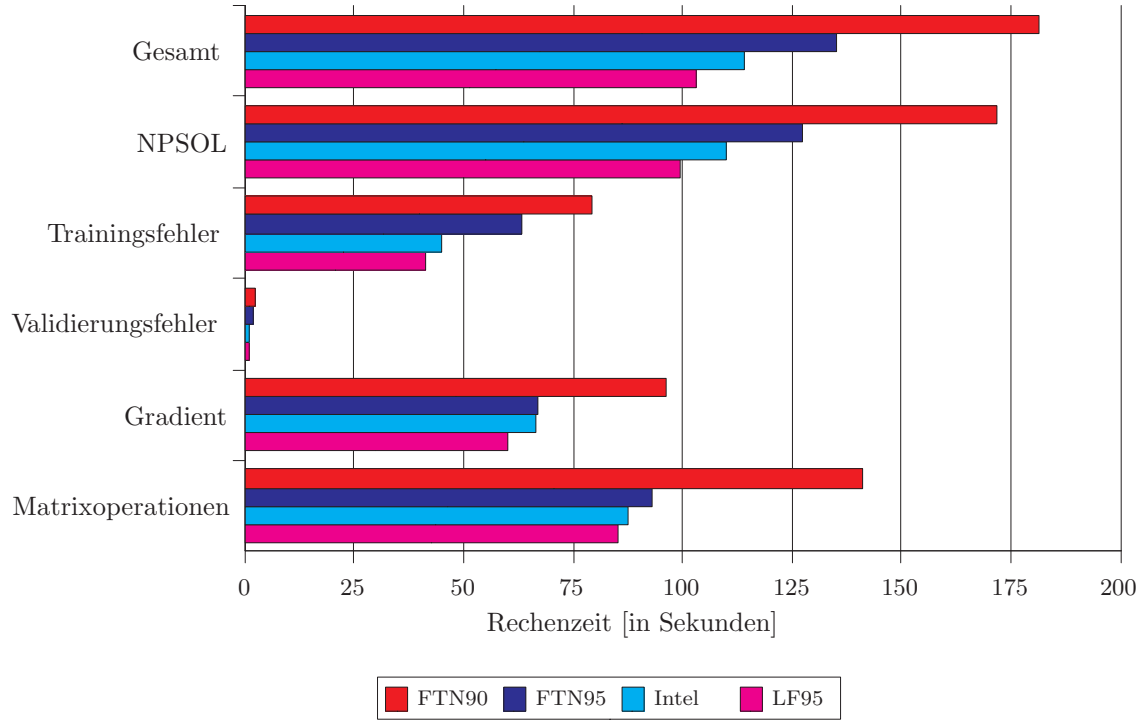


Abbildung G.11: Rechenzeiten des Ziffern-Beispiels (PENTIUM III)

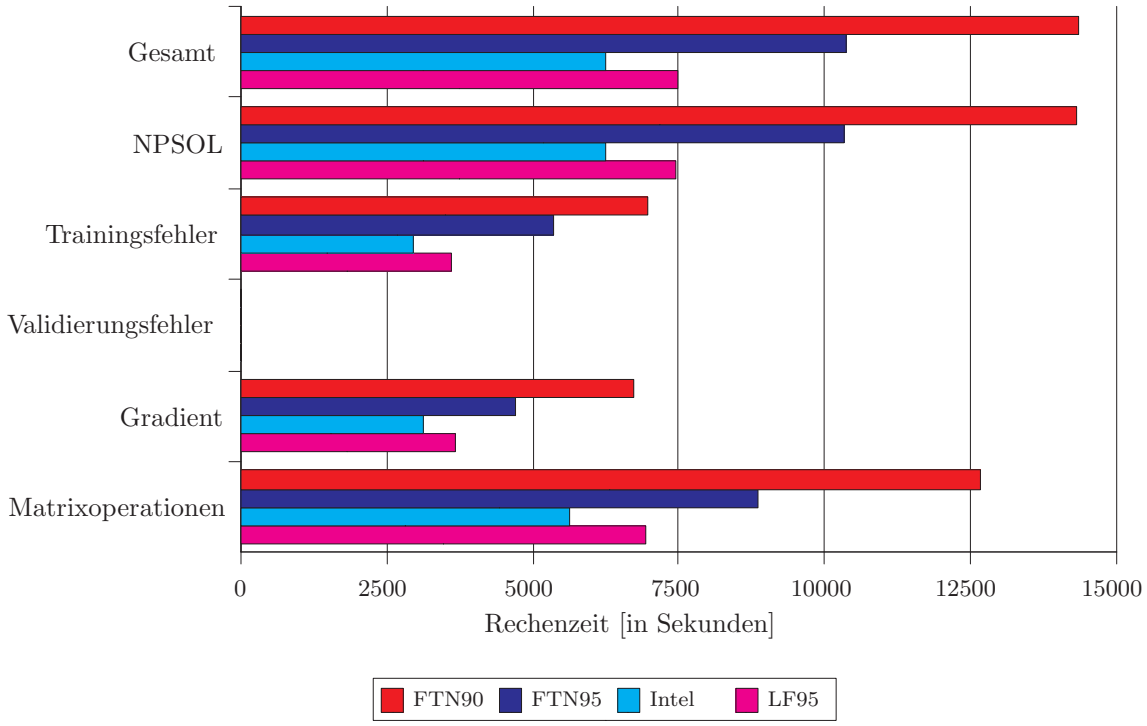


Abbildung G.12: Rechenzeiten des Zinsprognose-Beispiels (PENTIUM III)

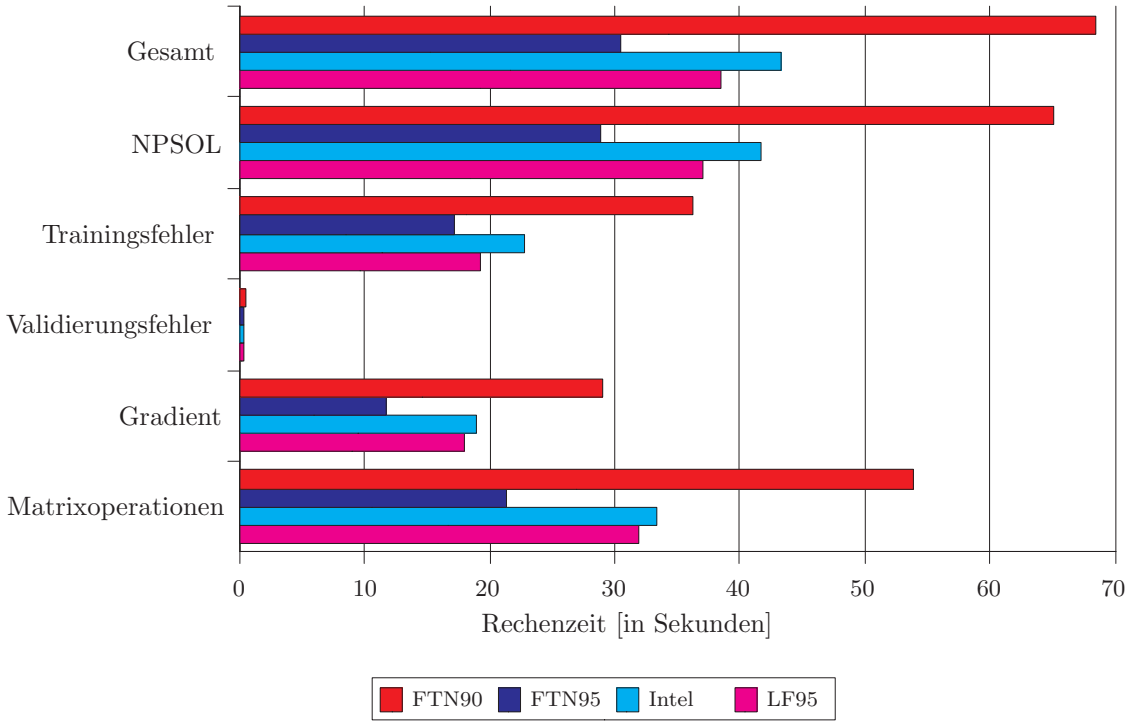


Abbildung G.13: Rechenzeiten des BASF Optionen-Beispiels (PENTIUM IV)

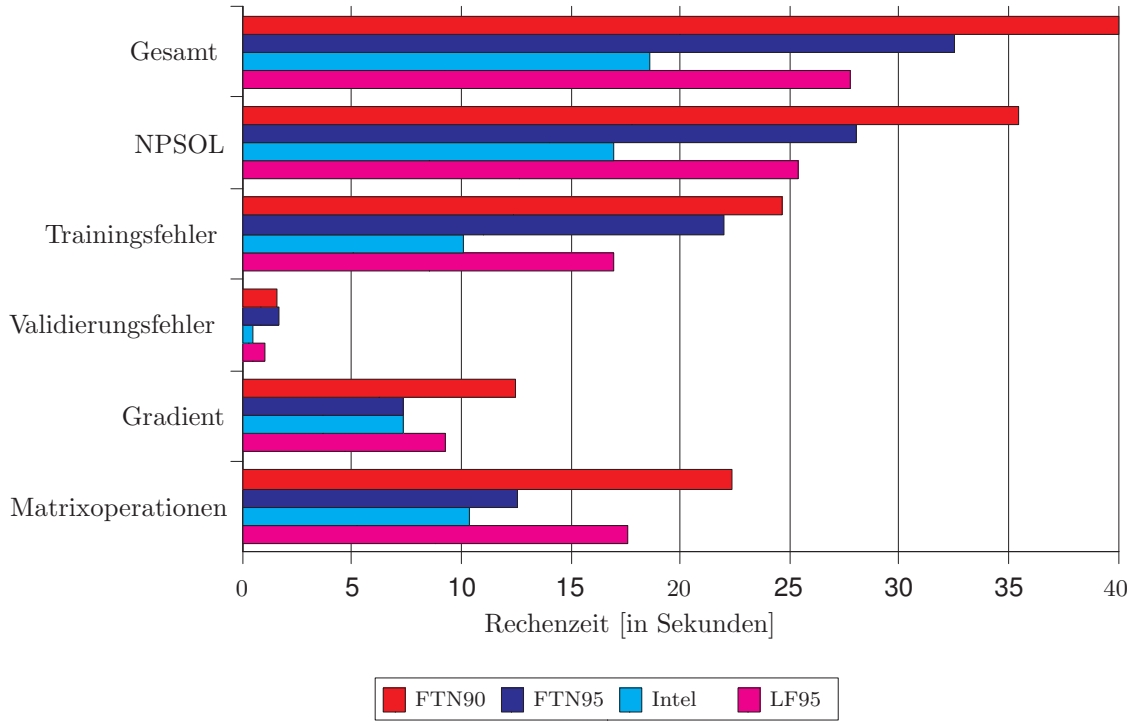


Abbildung G.14: Rechenzeiten des Euro-Dollar Optionen-Beispiels (PENTIUM IV)

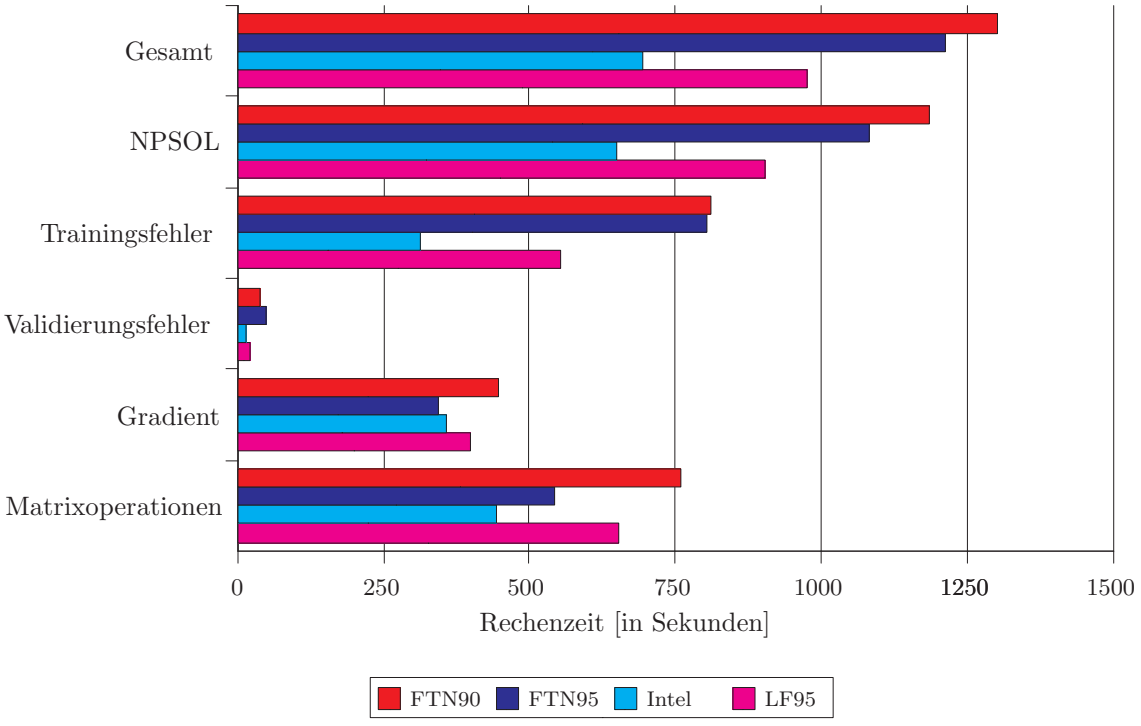


Abbildung G.15: Rechenzeiten des Pilze-Beispiels (PENTIUM IV)

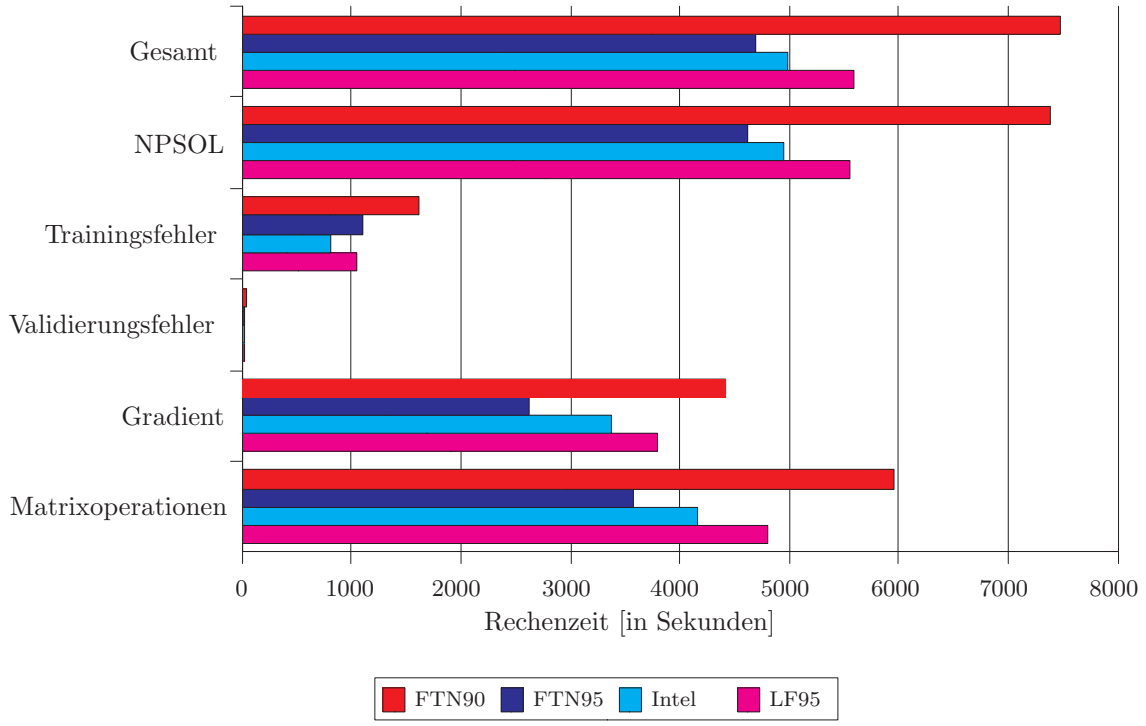


Abbildung G.16: Rechenzeiten des Space Shuttle Guidance-Beispiels (PENTIUM IV)

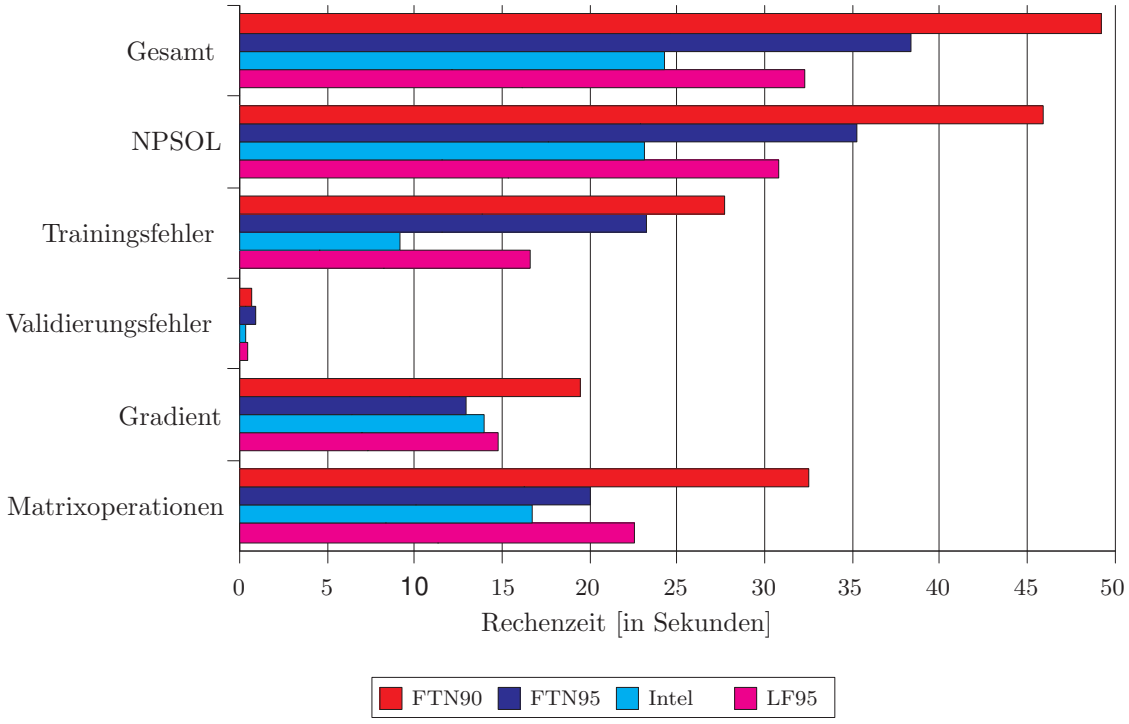


Abbildung G.17: Rechenzeiten des Ziffern-Beispiels (PENTIUM IV)

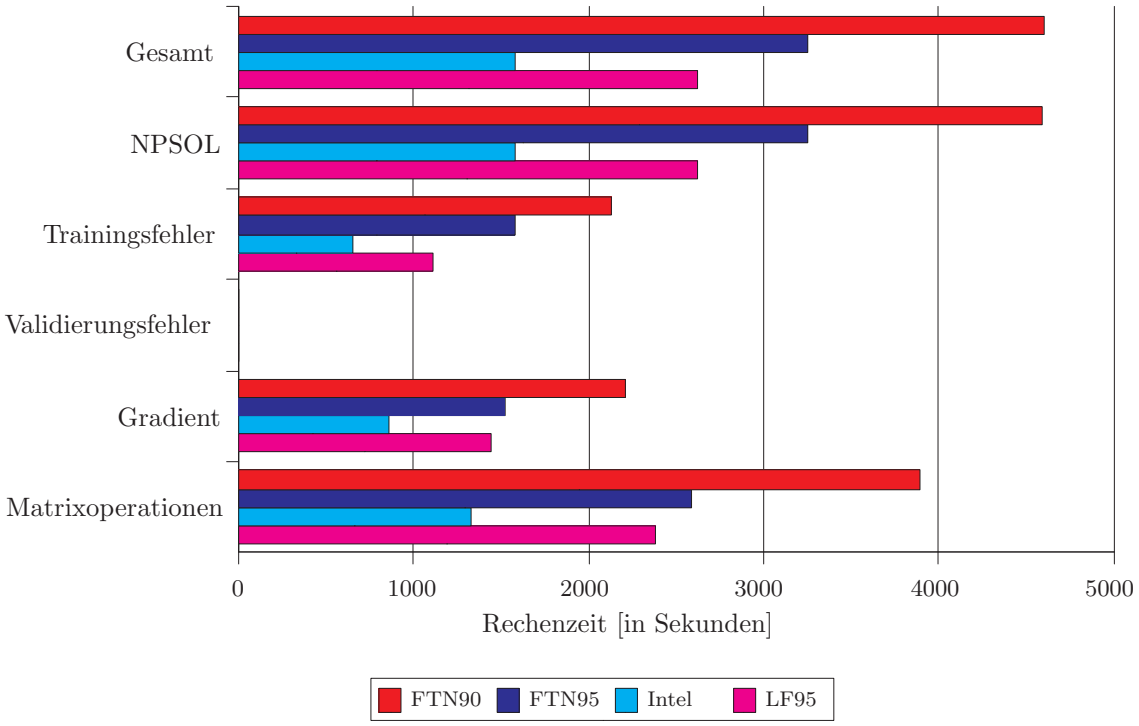
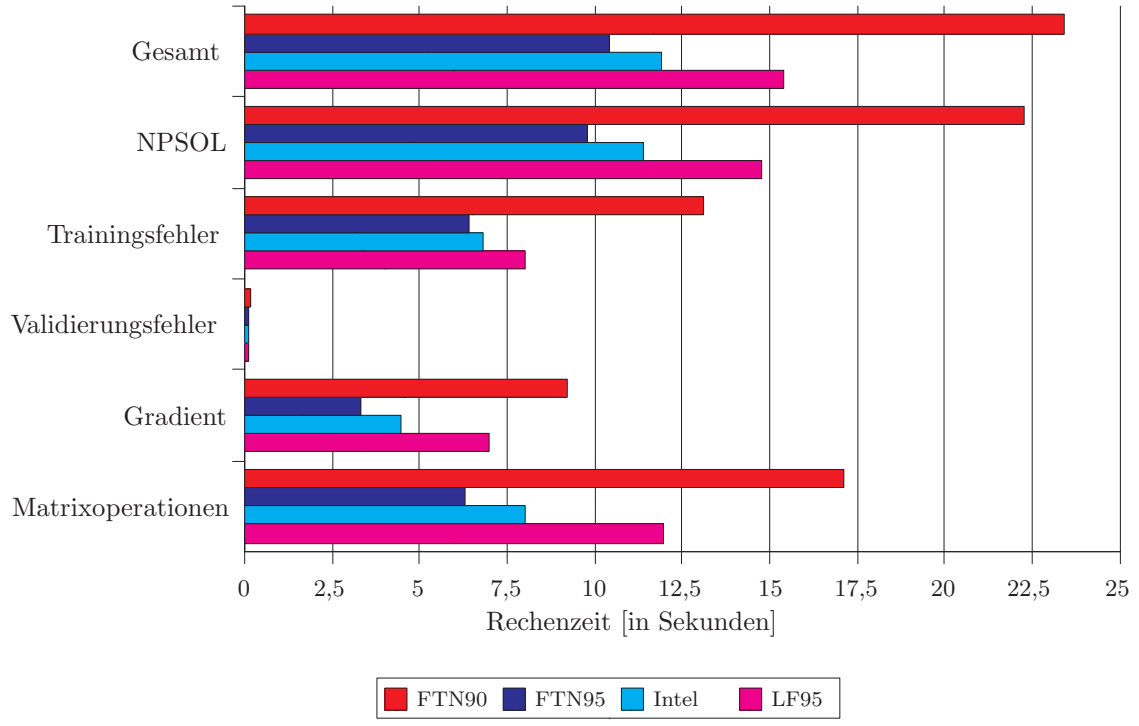


Abbildung G.18: Rechenzeiten des Zinsprognose-Beispiels (PENTIUM IV)

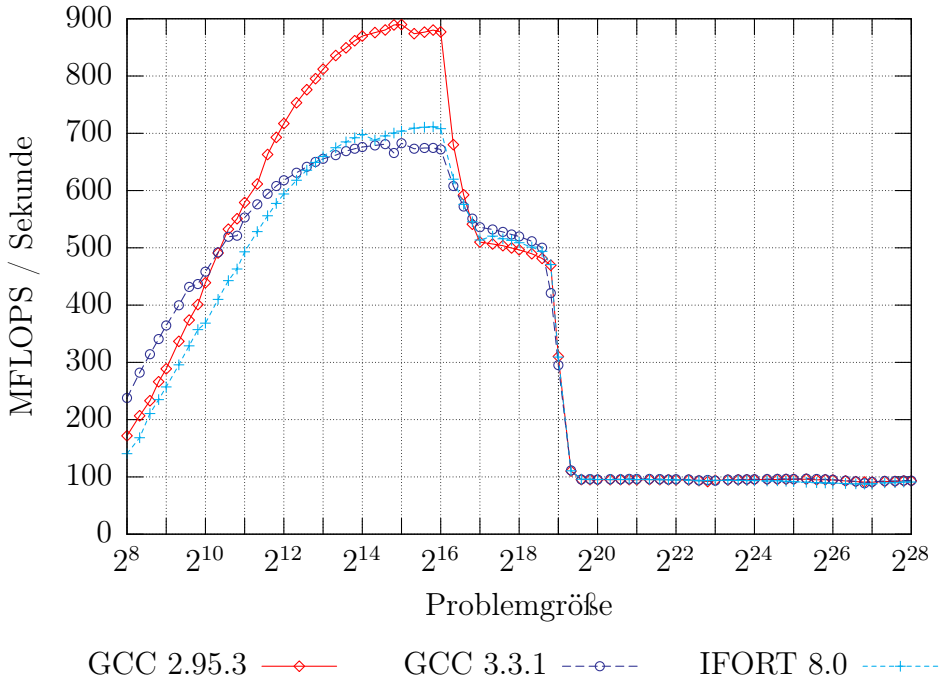


J.3 Installation unter Windows

Unter Windows ist zunächst die Unix-Emulation Cygwin zu installieren.[6] Natürlich müssen dabei auch die Pakete zur Entwicklung ausgewählt werden. Nach der Installation gibt es eine neue Verknüpfung auf dem Desktop. Sie öffnet einen Shell. Von da an ist wie unter Linux zu verfahren.[7]
Aus der ATLAS-Bibliothek kann auch eine dynamische Bibliothek für Windows erstellt werden. Dies ist besonders dann zu empfehlen, wenn ATLAS zusammen mit Windows-Compilern verwendet werden soll. Im Rahmen dieser Arbeit wurde dafür das Shell-Skript von Kevin Sheppard benutzt.[8] Es hat auch für die ACML und die Referenzimplementation gute Dienste geleistet.

J.4 Verwendung unterschiedlicher Compiler

Abbildung J.1: ATLAS-DAXPY-Leistungsvergleich (ATHLON XP-M / Linux)



[6] Das Installationsprogramm kann von der Homepage des Projekts unter `http://www.cygwin.com` heruntergeladen werden.
[7] Vgl. ATLAS (2003a), S. 15.
[8] Vgl. `http://www.kevinsheppard.com/research/MatlabAtlas/MatlabAtlas.htm`.

Abbildung J.2: ATLAS-DGEMV-Leistungsvergleich (ATHLON XP-M / Linux)

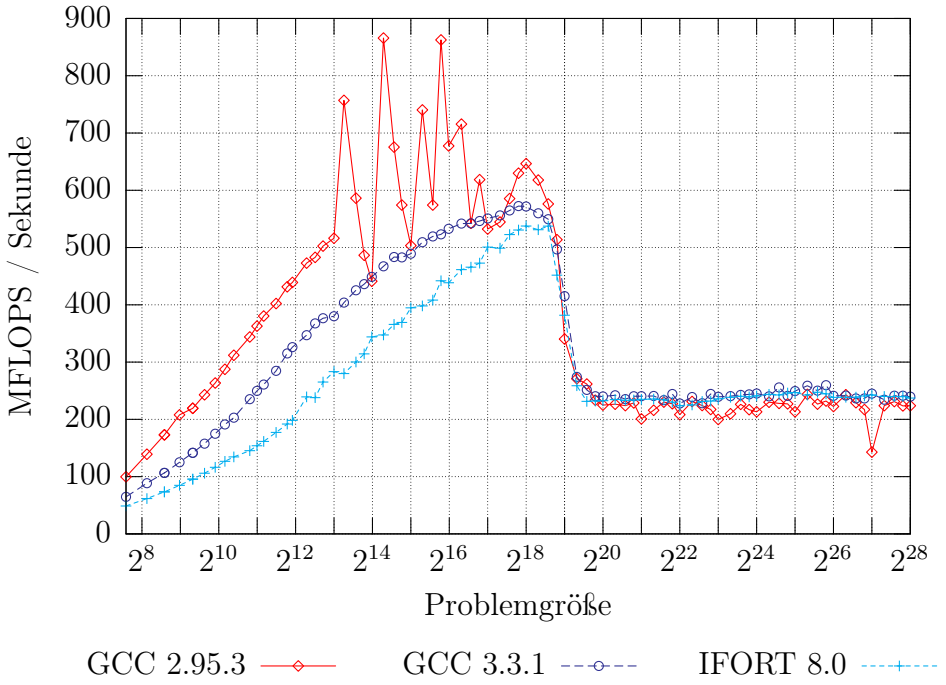


Abbildung J.3: ATLAS-DGEMM-Leistungsvergleich (ATHLON XP-M / Linux)