24. Juni 2025

FDS2 - Übung 9

SS 2025

Tim Peko

1. Beispiel 1: Zeitmessung von Grundoperationen

1.1. Lösungsansatz

Die Zeitmessung der grundlegenden Operationen `add`, `assign`, `compare`, `divide`, `index` und `multiply` erfolgte mit Hilfe der `pfc::timed_run` Funktion. Um trotz der begrenzten Timer-Auflösung im Millisekundenbereich genaue Messungen im Nanosekundenbereich zu erreichen, wurden folgende Maßnahmen ergriffen:

1.1.1. Messgenauigkeit

1. **Hochfrequente Wiederholung**: Jede Operation wird 100.000.000 Mal ausgeführt, um messbare Zeiten zu erhalten
2. **Compiler-Optimierung vermeiden**: Verwendung von `volatile` Variablen und explizite Ergebnisverwendung
3. **Störfaktoren eliminieren**: Isolierte Messung jeder Operation ohne I/O-Operationen während der Messung

1.1.2. Implementierungsdetails

Die Messungen erfolgen für folgende Operationen:

- **Add**: Integer-Addition zweier volatile Variablen
- **Assign**: Zuweisung zwischen zwei volatile Integer-Variablen
- **Compare**: Vergleichsoperation zwischen zwei volatile Integer-Variablen
- **Divide**: Gleitkomma-Division zweier volatile double-Variablen
- **Multiply**: Integer-Multiplikation zweier volatile Variablen
- **Index**: Zugriff auf zufällige Array-Elemente

Die Implementierung befindet sich in `example01\main01.cpp` und die Ergebnisse werden in eine CSV-Datei (`basic_operations_timing.csv`) exportiert für die Excel-Auswertung.

1.1.3. Hardware-Spezifikation

- **Betriebssystem**: Windows 10 (Build 26100)
- **Compiler**: Microsoft Visual C++ 1944
- **Timer-Auflösung**: Automatisch ermittelt via `pfc::get_timer_resolution()`
- **Prozessor**: x64 13th Gen Intel(R) Core(TM) i5-1335U @ 2.496GHz

1.2. Ergebnisse

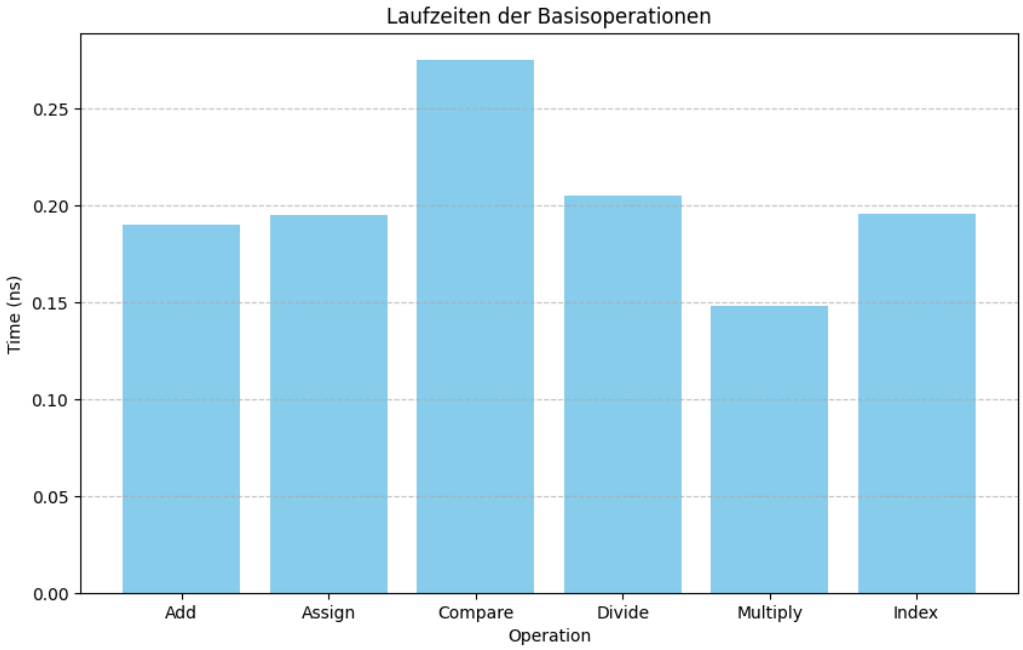

Abbildung 1: Laufzeit der Grundoperationen

2. Beispiel 2: Binäre Suche - Theorie vs. Praxis

2.1. Lösungsansatz

Der Vergleich zwischen theoretischer Analyse und praktischen Laufzeitmessungen erfolgt für drei verschiedene Implementierungen der binären Suche aus den Vorlesungsfolien:

2.1.1. Implementierte Algorithmen

1. **Binary Search V1**: Klassische Implementierung mit `left <= right` Bedingung
2. **Binary Search V2**: Optimierte Variante mit `left < right` und angepasster Mittelwert-Berechnung
3. **Binary Search V3**: Rekursive Implementierung

2.1.2. Testparameter

- **Array-Größen**: 1.000, 2.000, 4.000, 8.000, 16.000, 32.000, 64.000, 128.000 Elemente
- **Suchszenarien**:
  - Zufällig ausgewählte Werte aus dem Array (erfolgreich)
  - Nicht im Array enthaltene Werte (erfolglos)
- **Wiederholungen**: 1.000 Iterationen pro Messung für statistische Relevanz
- **Array-Belegung**: Sortierte gerade Zahlen (0, 2, 4, 6, …) für definierte „nicht gefunden“ Tests

2.1.3. Theoretische Analyse

Die Feinanalyse basiert auf den in Beispiel 1 ermittelten Grundoperations-Zeiten:

- **Vergleichsoperationen**: $\lceil \log_2(n) \rceil + 1$ pro Suche
- **Arithmetische Operationen**: Index-Berechnungen und Bereichs-Updates
- **Zuweisungen**: Variable Updates für `left`, `right`, `mid`

Die theoretischen Laufzeiten werden durch Multiplikation der Operationsanzahl mit den gemessenen Grundoperations-Zeiten berechnet.

Die Implementierung befindet sich in `example02\main02.cpp` als separates Visual Studio Projekt.

2.2. Testfälle

2.2.1. Korrektheitstests

Die Implementierung wurde mit einem Testarray `{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}` validiert:

Erfolgreich gefundene Werte:

```
Target 1: V1=0, V2=0, V3=0 - PASSED
Target 3: V1=1, V2=1, V3=1 - PASSED
Target 5: V1=2, V2=2, V3=2 - PASSED
[...weitere Tests...]
```

Nicht gefundene Werte:

```
Target 0: V1=-1, V2=-1, V3=-1 - PASSED
Target 2: V1=-1, V2=-1, V3=-1 - PASSED
Target 4: V1=-1, V2=-1, V3=-1 - PASSED
[...weitere Tests...]
```

Ergebnis: PASSED - Alle drei Implementierungen liefern korrekte Ergebnisse.

2.2.2. Zeitmessungen

Die systematischen Laufzeitmessungen werden für alle Kombinationen aus:

- 3 Algorithmus-Varianten
- 8 Array-Größen
- 2 Suchszenarien (gefunden/nicht gefunden)

durchgeführt und in `binary_search_analysis.csv` gespeichert.

2.2.3. Performance-Vergleich

Die Messungen ermöglichen den direkten Vergleich zwischen:

1. Berechneten theoretischen Laufzeiten basierend auf Operationsanzahl
2. Gemessenen praktischen Laufzeiten unter realen Bedingungen
3. Relative Performance der drei Algorithmus-Varianten

2.2.4. Störfaktor-Elimination

Um genaue Messungen zu gewährleisten:

- Verwendung von `volatile` für Ergebnisvariablen
- Zufällige Target-Generierung außerhalb der Zeitmessung
- Mehrfache Wiederholung für statistische Signifikanz
- Vermeidung von I/O-Operationen während der Messung

2.3. Ergebnisse und Auswertung

Die generierten CSV-Dateien ermöglichen eine detaillierte Excel-Analyse mit:

- Tabellarische Darstellung aller Messwerte
- Graphische Visualisierung der Laufzeitverläufe
- Vergleich zwischen Theorie und Praxis
- Performance-Ranking der Algorithmus-Varianten

Die Hardware-Spezifikation und Timer-Auflösung werden automatisch dokumentiert für die Reproduzierbarkeit der Ergebnisse.

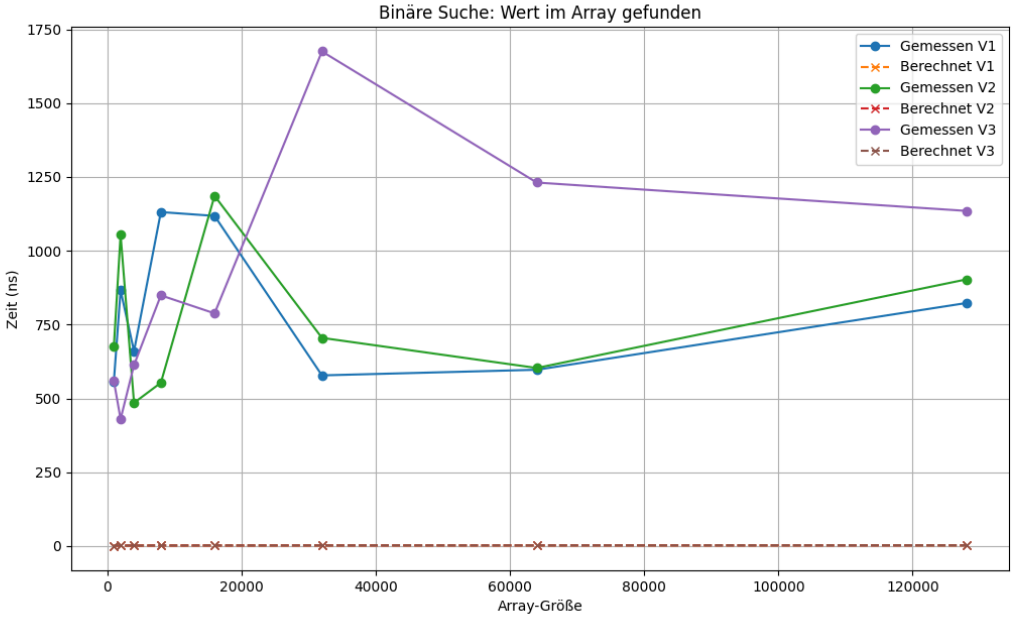

Abbildung 2: Laufzeit der Binärsuche (Werte gefunden)

Abbildung 3: Laufzeit der Binärsuche (Werte nicht auffindbar)

2.4. Projekt-Struktur

Die Lösung ist in zwei separate Visual Studio Projekte aufgeteilt:

2.4.1. Example01 - Grundoperationen

- Datei: `example01\main01.cpp`
- Zweck: Zeitmessung der 6 Grundoperationen (add, assign, compare, divide, multiply, index)
- Output: `basic_operations_timing.csv`
- Erweiterte System- und CPU-Informationen

- Reduzierte externe Abhängigkeiten (kein chrono)

2.4.2. Example02 - Binäre Suche

- Datei: `example02\main02.cpp`
- Zweck: Vergleich von 3 binären Suchvarianten (V1, V2, V3)
- Output: `binary_search_analysis.csv`
- Theoretische vs. praktische Laufzeitanalyse
- Umfassende Korrektheitstests und Edge-Case-Behandlung

Beide Projekte verwenden die `pfc-mini.hpp` Bibliothek für Zeitmessungen und sind in der `FDS_Peko_Ue09.sln` Solution enthalten.

Aufwand in h: 6