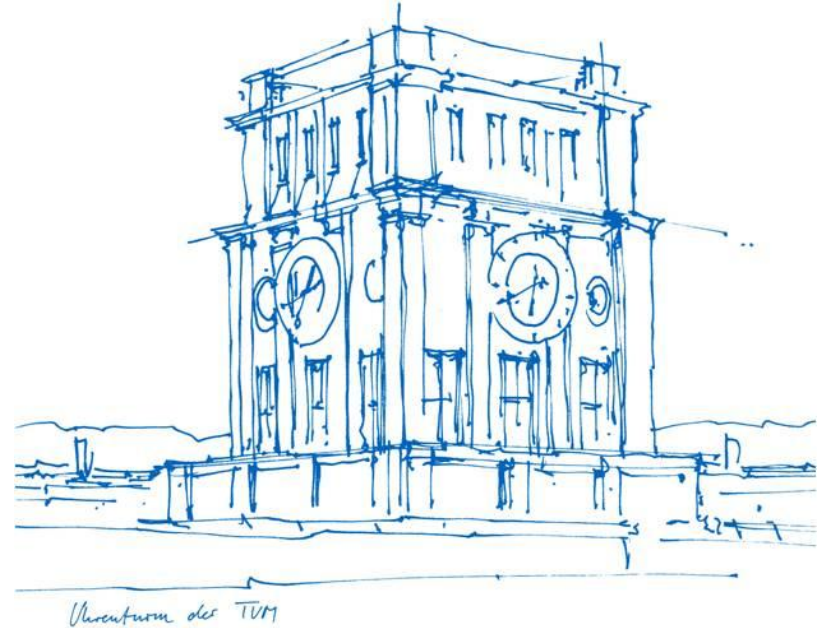


Memory Cache

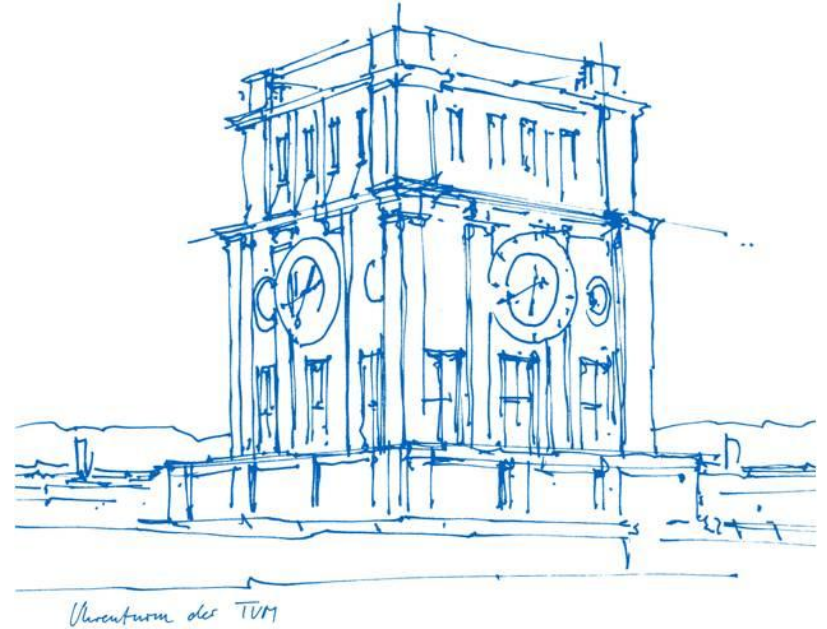
Roman Kupar, Lev Franko und Safie Emiramzaieva

München, 18.08.2025



1. Problemstellung
2. Lösungsansätze
 - Implementierung des SystemC-Modules
 - Integrierung in das Tiny-RISC CPU
 - Literaturrecherche
 - Rahmenprogramm
3. Korrektheit
 - Testen
 - Beispiele von Eingabedaten
4. Evaluation
 - Metriken
5. Zusammenfassung

Problemstellung



Bedarf nach Cache-Speicher

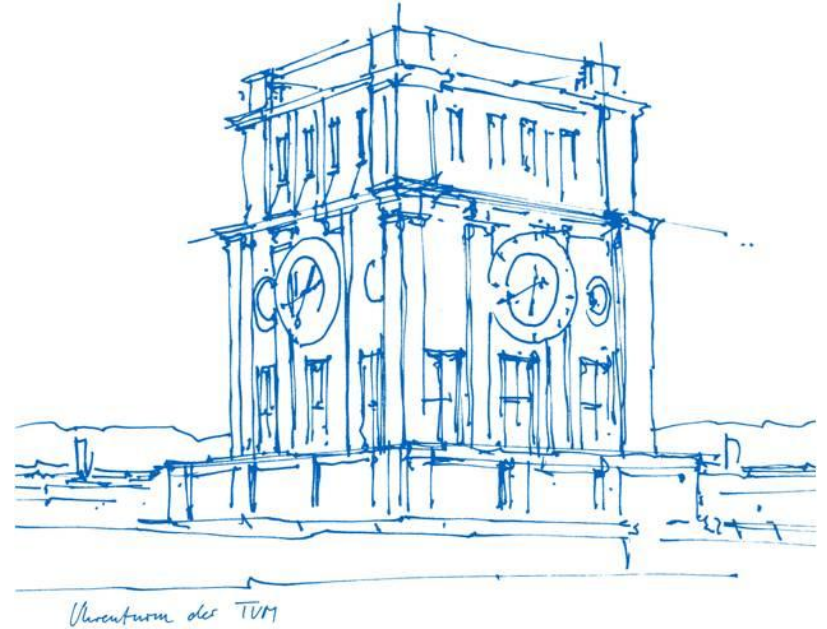
Motivation von Cache?

- Speichert häufig verwendete Daten im schnelleren und kleineren Speichertyp
- Reduziert den Flaschenhals des Hauptspeichers

Ziel des Projekts:

- Simulation eines Cache in SystemC
- Sammlung von Statistik (Misses, Hits)
- Vergleich verschiedener Zuordnungsstrategien
- Anhand sinnvoller Beispiele testen

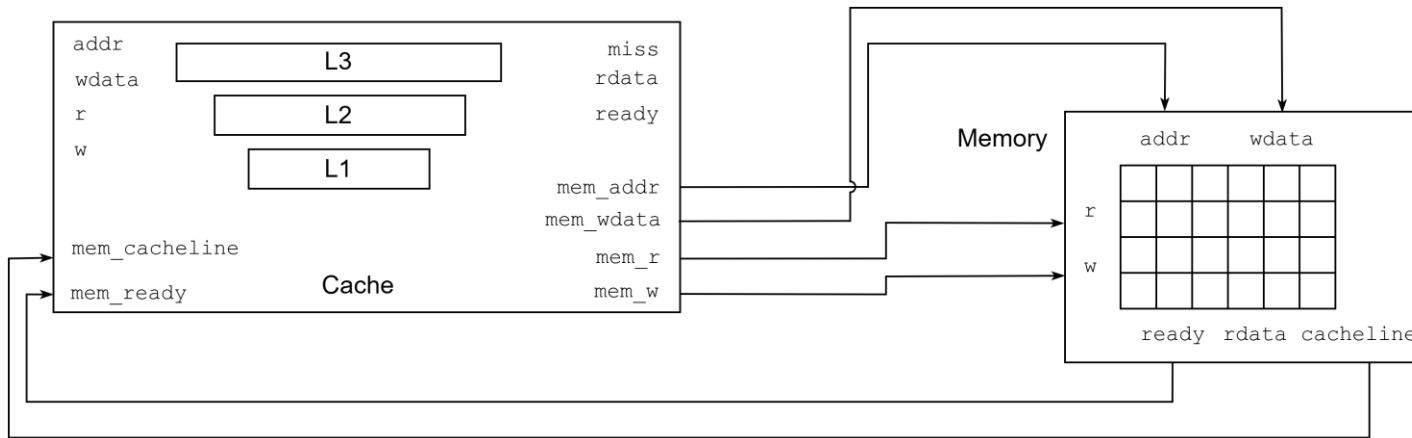
Lösungsansätze



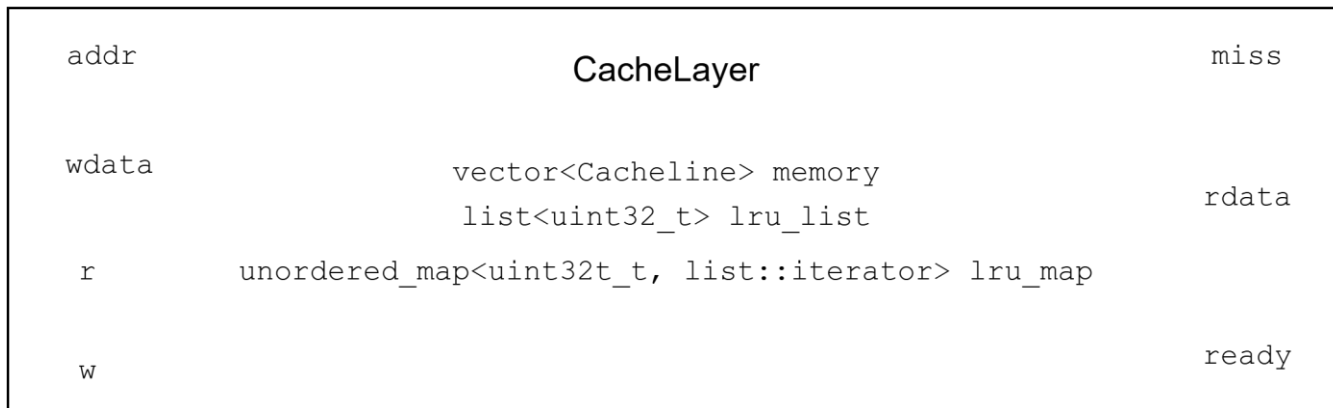
Der Aufbau des Cache-Memory

Cache-Memory setzt sich aus drei SystemC-Modulen zusammen:

- Cache
- CacheLayer
- MainMemory



- Steuert die Logik einer Cacheebene
- Cachezeile: Tag, Valid-Bit, Data (Vektor von Bytes)
- Implementierung von der LRU-Ersetzungsstrategie mittels Map und einer verketteten Liste
- Die verk. Liste speichert Indizes von Cachezeilen in LRU-Ordnung
- Map mit Verhältnis zwischen Tags und LRU-Indizes



Direct-Mapped:

- Offset, Index und Tag bestimmen
- Wenn Cachezeile valid und gespeicherte Tag stimmt
-> Hit

Fully-Associative:

- Offset und Tag bestimmen
- Wenn Tag in Map vorhanden ist -> Hit
- Index in der LRU-Liste nach vorne schieben

Bei Hit:

Lesen: Extraktion des Wortes aus der Cachezeile unter Zuhilfenahme des Offsets

Schreiben: das Wort in Cachezeile schreiben

Immer: Miss-Signal auf false setzen

Aufteilung einer Adresse:



Offset: untere Bits der Adresse als Index in die Zeile

Index: bestimmt die Cachezeile

Tag: beschreibt die Daten die abgelegt werden

Quelle: [ERA Vorlesung 7 "Caching", Folie 9](#)

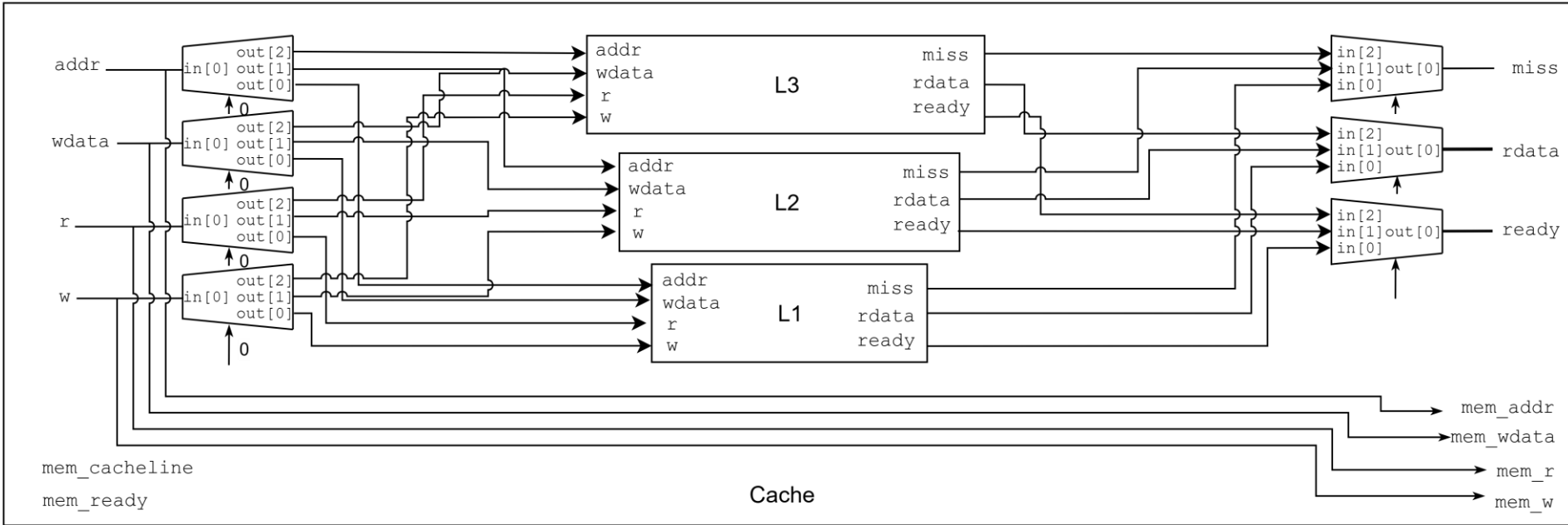
Lesezugriff im Cache

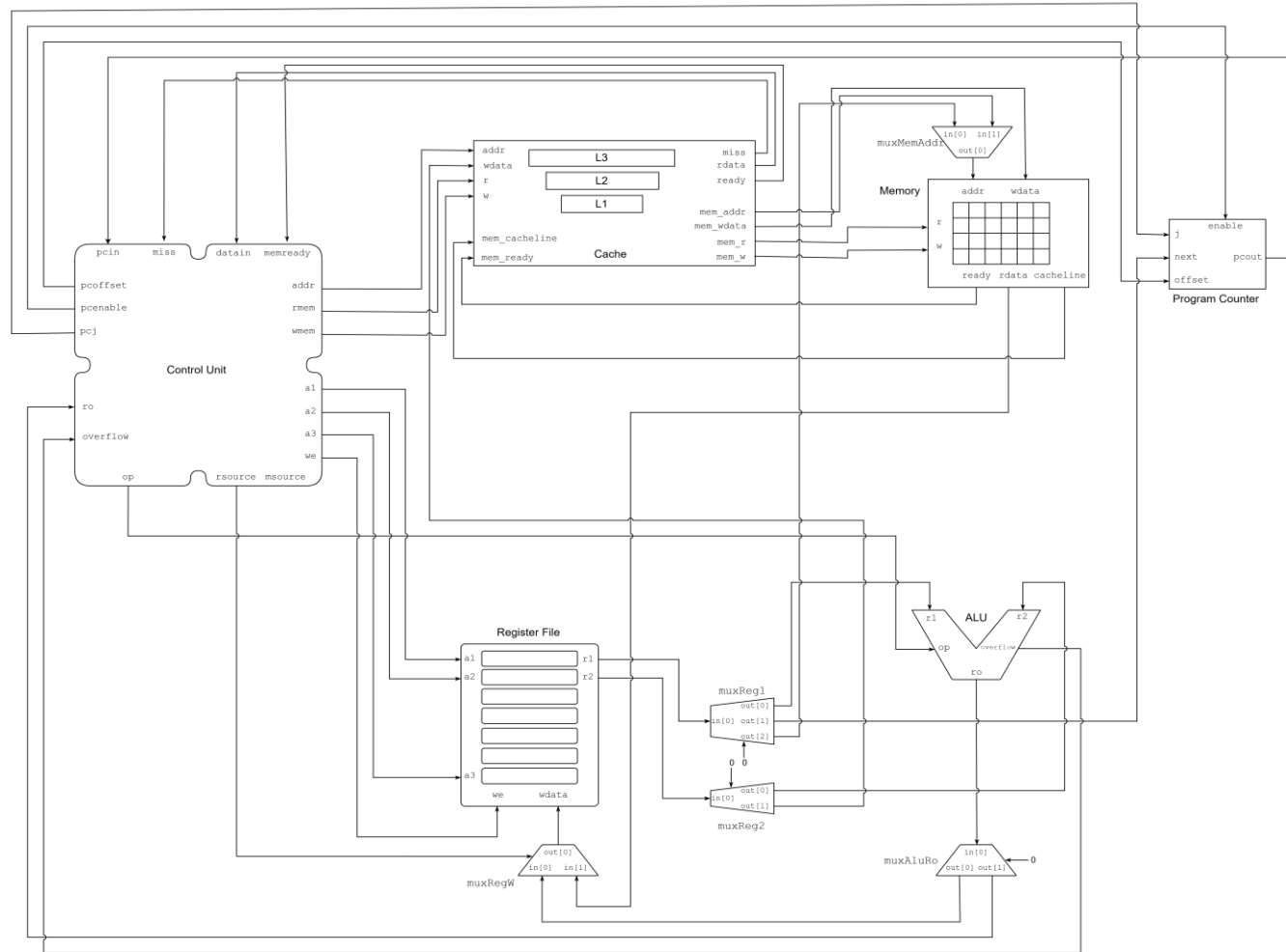
- Warten auf das Ready-Signal der jeweiligen Cacheebene
- Bei Hit: alle anderen Cacheebenen und den Hauptspeicher stoppen, Ausgabe-Signal weiterleiten
- Bei Miss: auf den Hauptspeicher warten, gelesene Cachezeile in alle Cacheebenen schreiben, gebrauchtes Wort aus der Cachezeile nehmen

Schreibzugriff im Cache

- Warten auf die jeweilige Cacheebene, Hit/Miss merken
- Warten auf den Hauptspeicher
- Cachezeile in Cacheebenen mit positivem Miss-Signal schreiben

Integration in TinyRISC CPU





Simulationsvorbereitung:

- Default-Werte bei fehlenden CLI-Parametern
Aufruf von `run_simulation(...)` in C++

Ausgabe:

- Parameter, Zyklen, Hits & Misses formatiert
ausgegeben und auch eine optionale Ausgabe
beim Debug-Modus

Fehlerbehandlung & Aufräumen:

- Sinnvolle Rückgabewerte bei Fehlern
Speicherfreigabe auch im Fehlerfall

Eingabedatei (CSV):

- Wird gepuffert, geprüft und in ein Request-Array
umgewandelt

CLI-Aufruf mit Argumenten:

`--cycles`, `--tf`, `--cacheline-size`, `--num-lines-l[1-3]`, `--latency-cache-l[1-3]`,
`--mapping-strategy`, `--debug`, `--test`

- Simulation mit benutzerfreundlichem Interface konfigurieren
- Mehrfache Ausführung von Simulationen mit unterschiedlichen Parametern – bequem und nacheinander
- Modularer Aufbau – funktioniert nur in Verbindung mit dem Rahmenprogramm, kein Ersatz für die Simulation



```
>Input file: requests.csv
Trace file: tracefile.tf
Cycles: 1000
Number of cache levels: 3
Cache line size: 64
Line number for L1 cache: 512
Line number for L2 cache: 4096
Line number for L3 cache: 32768
Latency L1: 8
Latency L2: 16
Latency L3: 32
Mapping strategy: Fully associative
Debug mode: OFF
Back
```

Typische Größen & Latenzen in modernen CPUs:

Größen:

- L1: 32-128KiB pro Kern.
- L2: 256KB-512KiB pro Kern.
- L3: 4-64MiB gemeinsam genutzt.

Cache-Linien-Größe: heute meist 64 Bytes.

Latenzen:

- L1: ca. 1–4 Zyklen.
- L2: 7–14 Zyklen
- L3: 20–40 Zyklen.

Unsere default values:

- **Cache-Linien-Größe** = 64
- **Größen:**
 - L1: 32KiB
 - L2: 256KiB
 - L3: 2MiB
- **Latenzen:**
 - L1: 8
 - L2: 16
 - L3: 32

LRU:

- + Gut bei temporärer Lokalität
- Overhead für Tracking, ineffizient bei Streaming

LFU:

- + Hält häufig genutzte Daten
- Träge bei Musterwechsel; hoher Verwaltungsaufwand

FIFO:

- + Einfache Umsetzung
- Ignoriert Nutzungsmuster, meist schlechtere Trefferquote

Random Replacement:

- + Sehr geringer Overhead, simpel
- Zufällig, geringe Vorhersagbarkeit

Speicherzugriffsverhalten eines speicherintensiven Algorithmus

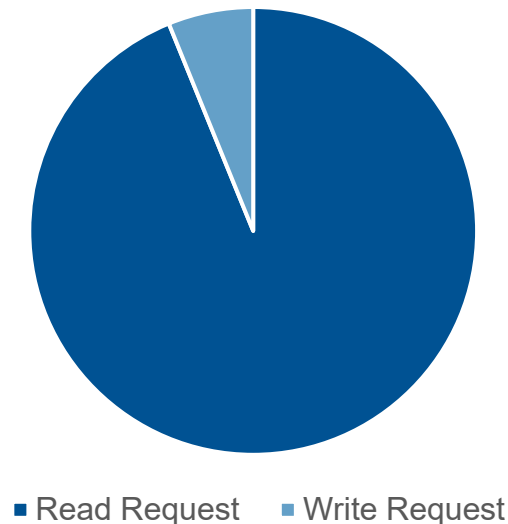
Unsere Wahl: Matrixmultiplikation

- Macht zahlreiche Lese- und Schreibzugriffe
- Gut zur Untersuchung von Speicherverhalten
- Python-Skript generiert Requests -> Test der gesamten Simulation

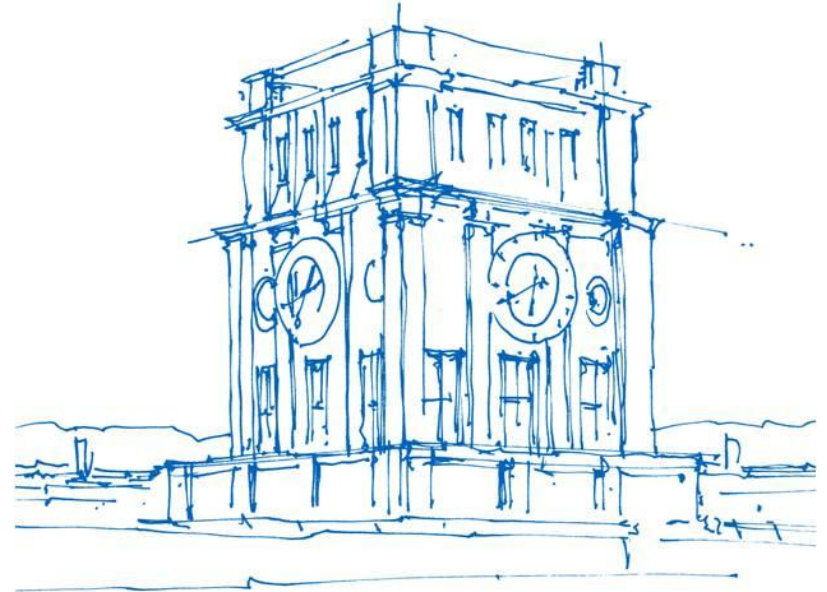
Beispiel 30x30 Matrixmultiplikation:

- 58500 Requests -> 3600 Schreib- und 54900 Lesezugriffe
- 916200 Taktzyklen mit 99,7% Hit-Rate;
539% schneller als ohne Cache

Proportion Read and Write Requests



Korrektheit



Uhrenturm der TUM





Automatisierte Tests

Das Testen dieses Programms ist automatisiert durch ein **Python-Skript**.

Tests starten mit dem Make-Flag `make run-unit-tests`.

Abdeckung von relevanten Edge Cases im Rahmen des Programms:

- Ungültige Optionen (CLI)
- Falsche Argumente wie Literale, negative Werte oder nicht erlaubte Werte
- Fehlerhafte oder ungültige CSV-Dateien
- Edge Cases bei der Validierung der Eingabe
- Fehlerbehandlung bei Dateien

Directory	Line Coverage ↕		Functions ↕	
/home/roman/GRA/C-Aufgaben/gra25sproject-t141/include		87.6 %	369 / 421	97.1 % 33 / 34
/home/roman/GRA/C-Aufgaben/gra25sproject-t141/src		92.0 %	195 / 212	75.0 % 3 / 4
/home/roman/GRA/C-Aufgaben/gra25sproject-t141/src/parsers		93.6 %	160 / 171	100.0 % 8 / 8
/home/roman/GRA/C-Aufgaben/gra25sproject-t141/util		75.9 %	41 / 54	100.0 % 5 / 5

Die Abdeckungsdaten wurden automatisch mit [LCOV](#) generiert

Einfaches Beispiel

Schreiben und sofort
aus L1 lesen:

- W,0x0010,20
- R,0x0010,

SIMULATION: Request 1: type=W, addr = 0x00000010, data=0x00000014

CACHE_LAYER 1: Cache Memory Content:

Index	Tag	Valid	Data
0	2	true	14 0 0 0 0 0 0 0

CACHE_LAYER 2: Cache Memory Content:

Index	Tag	Valid	Data
0	2	true	14 0 0 0 0 0 0 0

CACHE_LAYER 3: Cache Memory Content:

Index	Tag	Valid	Data
0	2	true	14 0 0 0 0 0 0 0

SIMULATION: Request 2: type=R, addr = 0x00000010, data=0x00000014

MAIN: Read data in CACHE_LAYER[1]: 20

SIMULATION: Read data: 20

Beispiel Hit in L2

SIMULATION: Request 17: type=R, addr = 0x00000020, data = 0x00000028

MAIN: Hit in L[2]: true

SIMULATION: Read data: 637534248

CACHE_LAYER 1: Cache Memory Content:

Index	Tag	Valid	Data
0	2	true	14 0 0 0 0 0 0 0
1	3176	true	0 0 0 23 e9 b6 58 0
2	1128	true	0 0 50 17 df 8 0 0
3	9766	true	0 0 0 0 23 e9 b6 58
4	2190	true	0 0 23 e9 b6 58 0 0
5	614	true	0 0 23 e9 b6 58 0 0
6	594	true	0 0 23 e9 b6 58 0 0
7	9764	true	0 0 0 23 e9 b6 58 0

CACHE_LAYER 2: Cache Memory Content:

Index	Tag	Valid	Data
0	2	true	14 0 0 0 0 0 0 0
1	4	true	28 0 0 26 9 0 0 0
2	1128	true	0 0 50 17 df 8 0 0
3	9766	true	0 0 0 0 23 e9 b6 58
4	2190	true	0 0 23 e9 b6 58 0 0
5	614	true	0 0 23 e9 b6 58 0 0
6	594	true	0 0 23 e9 b6 58 0 0
7	9764	true	0 0 0 23 e9 b6 58 0
8	3176	true	0 0 0 23 e9 b6 58 0

Beispiel LRU Ersetzungsstrategie

SIMULATION: Request 16: type=W, addr = 0x00006343, data=0x08888888

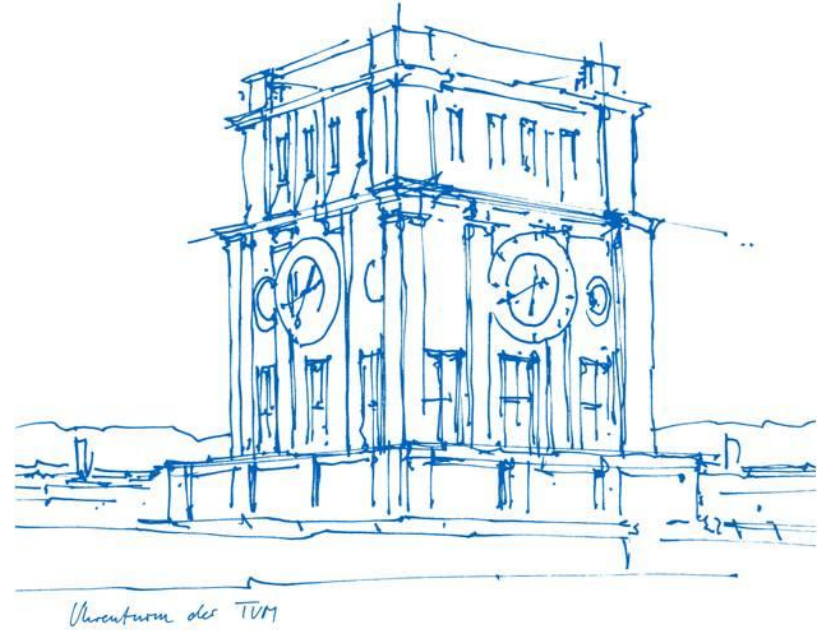
CACHE_LAYER 1: Cache Memory Content:

Index	Tag	Valid	Data
0	2	true	14 0 0 0 0 0 0 0
1	4	true	28 0 0 26 9 0 0 0
2	1128	true	0 0 50 17 df 8 0 0
3	9766	true	0 0 0 0 23 e9 b6 58
4	2190	true	0 0 23 e9 b6 58 0 0
5	614	true	0 0 23 e9 b6 58 0 0
6	594	true	0 0 23 e9 b6 58 0 0
7	9764	true	0 0 0 23 e9 b6 58 0

CACHE_LAYER 1: Cache Memory Content:

Index	Tag	Valid	Data
0	2	true	14 0 0 0 0 0 0 0
1	3176	true	0 0 0 88 88 88 8 0
2	1128	true	0 0 50 17 df 8 0 0
3	9766	true	0 0 0 0 23 e9 b6 58
4	2190	true	0 0 23 e9 b6 58 0 0
5	614	true	0 0 23 e9 b6 58 0 0
6	594	true	0 0 23 e9 b6 58 0 0
7	9764	true	0 0 0 23 e9 b6 58 0

Evaluation



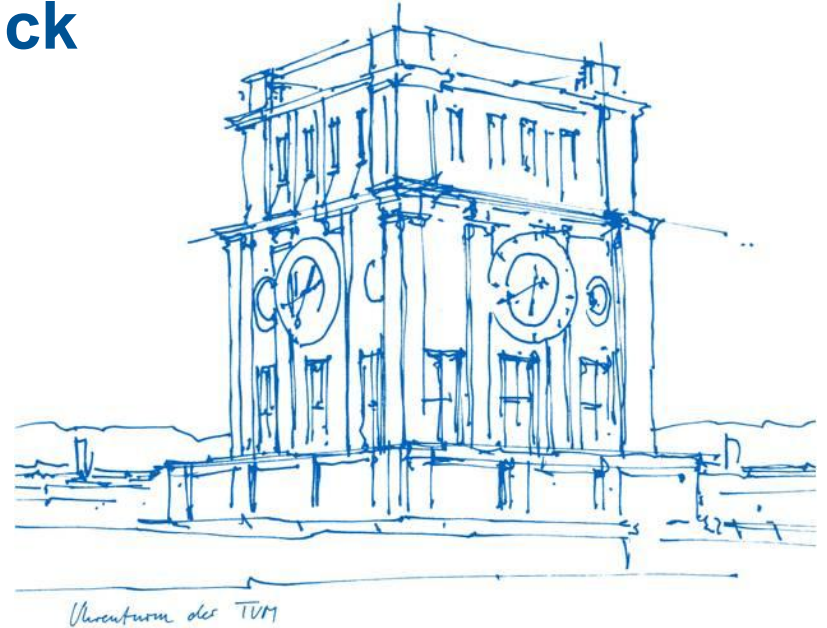
Genutzte Metriken:

- Hit-Rate: bei groß genug Cache konvergiert zu 100%
- Anzahl an gebrauchten Taktzyklen
- Vergleich mit der Simulation ohne Cache (nur MainMemory)

Vergleich Direct-Mapped vs. Fully-Associative Mapping:

- Matrixmultiplikation 20x20: DM hat mehr Misses als FA (1068 bei DM gegen 337 bei FA)
-> Senkung von Hit-Rate bzw. Effizienz

Zusammenfassung und Ausblick



Was haben wir beobachtet?

- Cache erhöht Zugriffssperformance deutlich bei räumlicher Lokalität
- Cache-Misses verursachen merkliche Verzögerung (Main-Memory-Zugriff)
- Zugriffsmuster wie bei Matrixmultiplikation verdeutlichen Schwächen & Optimierungspotenziale

Was wurde erreicht ?

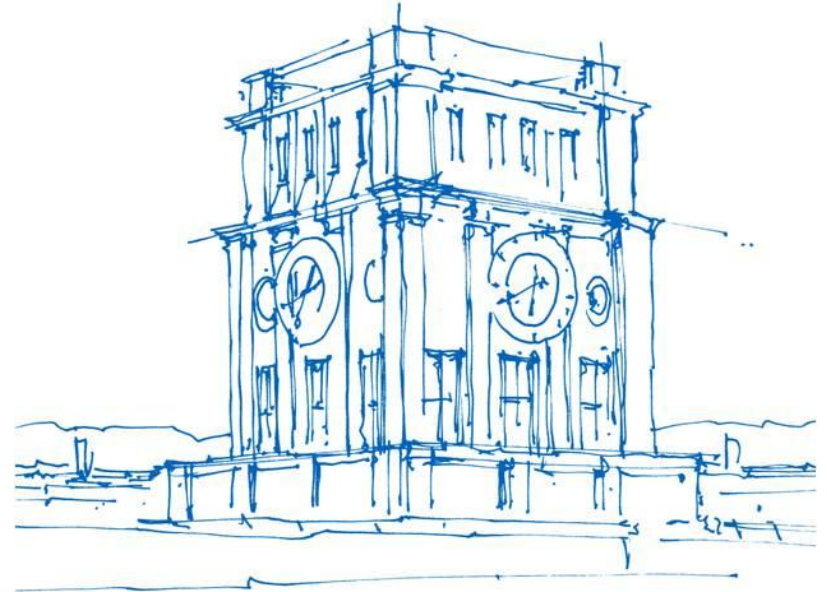
- Implementierung eines hierarchischen Cache
- Realistische Simulation des Speicherzugriffsverhaltens mit:
 - Sammlung der Statistik
 - Kommunikation mit dem Hauptspeicher
- Unterstützung verschiedener Parameter
- Alle Tests bestanden

Mögliche Verbesserungen



- Erweiterung mit Set-Associative Mapping-Strategie
- Einführung von Write-Back
- Smartere Ersetzungsstrategie (Kombination von LRU und LFU)

Jetzt Fragerunde!



Uhrenturm der TUM