

# Cache Simulation and Analyse

Translation Lookaside Buffer

Paritosh Deshmukh, Altay Özkan,  
Shaurya Songara



# Problemstellung:

- Untersuchung des Einflusses von TLBs auf die Ausführungszeit von Speicherzugriffen.
- Relevanz von TLBs in modernen Prozessorarchitekturen.



# TLBs und ihre Bedeutung

- Definition von TLBs:
  - Schnelle Speicher, die Seitenübersetzungen zwischenspeichern.
- Funktion von TLBs:
  - Reduzierung direkter Speicherzugriffe.
  - Verbesserung der Systemleistung.



# Ziele des Projekts

- Hauptfragestellungen:

- Einfluss von TLB-Größen und -Architekturen.

- Auswirkungen der Latenzen auf die Systemleistung.

- Optimierungspotential von TLBs.

- Erwartete Vorteile:

- Verbesserte Systemleistung durch effiziente TLB-Nutzung.



# Detaillierte Spezifikation

- TLB-Größen und -Architekturen:
  - Typische Größen und ihre Auswirkungen.
  - Verschiedene Architekturen und ihre Effizienz.
- Speicher- und TLB-Latenzen:
  - Vergleich der Latenzen von Hauptspeicher und TLBs.



# Analyse der Problemstellung

- Speicherzugriffsverhalten:
  - Untersuchung der Speicherzugriffe bei einer verketteten Liste.
  - Erstellung und Nutzung von CSV-Dateien zur Simulation.



# Vorgehensweise

- Schritte zur Untersuchung:
  - Recherche und Dokumentation.
  - Simulation und Analyse.
  - Bewertung der Ergebnisse.
- Methodik:
  - Simulation verschiedener TLB-Konfigurationen.
  - Vergleich von Trefferraten und Latenzen.

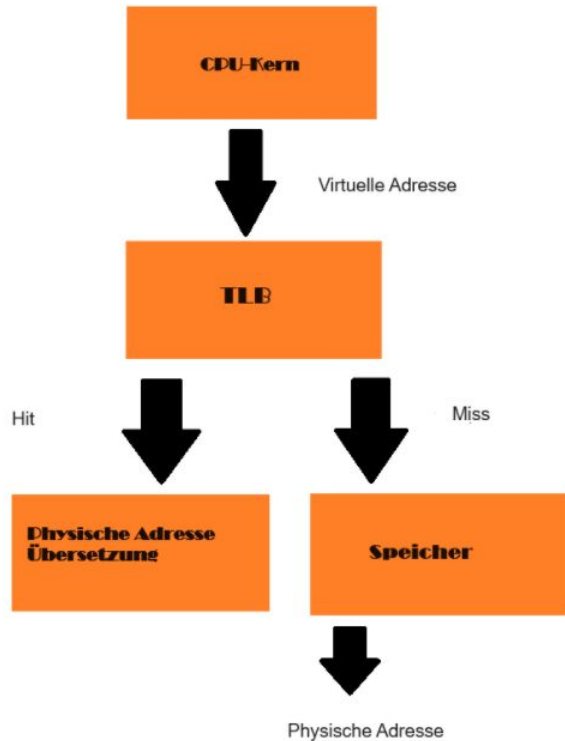


# Lösungsansätze und Optimierungen

## Systemarchitektur

- TLB-Modul:
  - Speichert und verwaltet die Adressübersetzungen.
  - Empfängt virtuelle Adressen vom CPU-Kern.
  - Gibt entweder die entsprechende physische Adresse zurück (Treffer) oder signalisiert einen Fehlertreffer (Miss).
  - Bei einem Fehlertreffer wird auf den Hauptspeicher zugegriffen, um die physische Adresse zu berechnen und das TLB zu aktualisieren.
- Simulationsmodul:
  - Führt die Speicherzugriffe basierend auf den Anfragen in der CSV-Datei aus.
  - Überwacht die Zyklenanzahl, Treffer und Fehlertreffer.
  - Aktualisiert die Ergebnisse entsprechend.





```
void tlb_lookup() {  
    if (reset.read() == true) {  
        physicalAddr.write(0);  
        hit.write(false);  
        return;  
    }  
  
    uint32_t virtualAddr_read = virtualAddr.read();  
    uint32_t index = (virtualAddr_read >> offset_bits) & ((1 << index_bits) - 1); // Extract the index  
    uint32_t tag = virtualAddr_read >> (offset_bits + index_bits); // Extract the tag  
  
    if (tlb[index].valid && tlb[index].tag == tag) {  
        physicalAddr.write(tlb[index].physicalAddr + (virtualAddr_read & ((1 << offset_bits) - 1))); // Combine physical page and offset  
        hit.write(true);  
    } else {  
        hit.write(false);  
    }  
}
```

```
void tlb_update(uint32_t virtAddr, uint32_t physAddr) {  
    uint32_t index = (virtAddr >> offset_bits) & ((1 << index_bits) - 1); // Extract the index  
    uint32_t tag = virtAddr >> (offset_bits + index_bits); // Extract the tag  
    tlb[index].tag = tag;  
    tlb[index].physicalAddr = physAddr & ~((1 << offset_bits) - 1); // Store the physical page number  
    tlb[index].valid = true;  
}
```



# Algorithmen und Implementierung

TLB-Zugriff:

- Lese die virtuelle Adresse und extrahiere den Index sowie das Tag.
- Überprüfe, ob der TLB-Eintrag gültig ist und das Tag übereinstimmt.
- Bei einem Treffer wird die physische Adresse zurückgegeben.

Fehlertreffer und Aktualisierung:

- Bei einem Fehlertreffer wird die physische Adresse durch Zugriff auf den Hauptspeicher berechnet.
- Das TLB wird mit dem neuen Eintrag aktualisiert.
- Die physische Adresse wird zurückgegeben.

Start-> Lese die Virtuelle Adresse -> Suche in der TLB -> Ist die virtuelle Adresse in der TLB->(Ja?)->(Gebe die physische Adresse zurück)->(Ende)->Nein->Greife auf den Speicher zu-> Berechne die physische Adresse -> Aktualisiere die TLB -> Gebe die physische Adresse zurück -> Ende



# Optimierungen

- Effiziente Adressübersetzung:
  - TLB in einem Vektor gespeichert für schnellen Zugriff auf die Einträge.(Anstatt Liste)
- Speichermanagement:
  - Effizientes Speichermanagement zur Vermeidung unnötiger Zugriffe.
  - Beschleunigung der Simulation.



# Genauigkeit und Korrektheit des Projekts

Häufige Verwendung von Anweisungen, die gedruckt wurden, um zu bestätigen, dass der Code ordnungsgemäß funktioniert.

```
std::cout << "Simulation started." << std::endl;
```

```
std::cout << "Cycle: " << result.cycles << ", Current Request: " << currentRequest << std::endl;
```

```
std::cout << "Simulation complete" << std::endl;
```

```
std::cout << "Trace file opened successfully." << std::endl;
```

Folglich können wir bestätigen, dass der Kontrollfluss des Codes so ist, wie er sein muss.

die Ergebnisse der Testläufe und die entsprechenden Tracefiles sind ebenfalls zufriedenstellend, wobei eine höhere tlb-Größe zu einer geringeren Anzahl von Zyklen führt.


```
Cycle: 450, Current Request: 45
Cycle: 482, Current Request: 46
Simulation complete
```

```
Info: /OSCI/SystemC: Simulation stopped by user.
Simulation finished.
Cycles: 482, Hits: 33, Misses: 13
Cycles: 482
Hits: 33
Misses: 13
Primitive Gate Count: 14120
```

```
Miss: Virtual Address 1f2, Translated Physical Address 232
Hit: Virtual Address 1f2, Physical Address 232
Miss: Virtual Address 2a0, Translated Physical Address 2e0
Hit: Virtual Address 2a0, Physical Address 2e0
Miss: Virtual Address 300, Translated Physical Address 340
Hit: Virtual Address 300, Physical Address 340
Miss: Virtual Address 450, Translated Physical Address 490
Hit: Virtual Address 450, Physical Address 490
Miss: Virtual Address 500, Translated Physical Address 540
Hit: Virtual Address 500, Physical Address 540
Miss: Virtual Address 600, Translated Physical Address 6a0
Hit: Virtual Address 600, Physical Address 6a0
Miss: Virtual Address 700, Translated Physical Address 740
Hit: Virtual Address 700, Physical Address 740
Miss: Virtual Address 800, Translated Physical Address 840
Hit: Virtual Address 800, Physical Address 840
Miss: Virtual Address 900, Translated Physical Address 940
Hit: Virtual Address 900, Physical Address 940
Miss: Virtual Address a00, Translated Physical Address a40
Hit: Virtual Address a00, Physical Address a40
Miss: Virtual Address b00, Translated Physical Address b40
Hit: Virtual Address b00, Physical Address b40
Miss: Virtual Address c00, Translated Physical Address c40
Hit: Virtual Address c00, Physical Address c40
Miss: Virtual Address d00, Translated Physical Address d40
Hit: Virtual Address d00, Physical Address d40
Miss: Virtual Address e00, Translated Physical Address e40
Hit: Virtual Address e00, Physical Address e40
Miss: Virtual Address f00, Translated Physical Address f40
Hit: Virtual Address f00, Physical Address f40
Miss: Virtual Address 1100, Translated Physical Address 1140
Hit: Virtual Address 1100, Physical Address 1140
Miss: Virtual Address 1200, Translated Physical Address 1240
Hit: Virtual Address 1200, Physical Address 1240
Miss: Virtual Address 1300, Translated Physical Address 1340
Hit: Virtual Address 1300, Physical Address 1340
Miss: Virtual Address 1400, Translated Physical Address 1440
Hit: Virtual Address 1400, Physical Address 1440
Miss: Virtual Address 1500, Translated Physical Address 1540
Hit: Virtual Address 1500, Physical Address 1540
Miss: Virtual Address 1600, Translated Physical Address 1640
Hit: Virtual Address 1600, Physical Address 1640
Miss: Virtual Address 1700, Translated Physical Address 1740
Hit: Virtual Address 1700, Physical Address 1740
Miss: Virtual Address 1800, Translated Physical Address 1840
Hit: Virtual Address 1800, Physical Address 1840
Miss: Virtual Address 1900, Translated Physical Address 1940
Hit: Virtual Address 1900, Physical Address 1940
```

```
go34mik@Wakul:~/gra24cdaproject-g162$ ./main -c 1000 --blocksize 64 --v2b-block-offset 2 --tlb-size 32 --tlb-latency 2
-memory-latency 30 Input.csv.csv
run_simulation called with parameters:
Cycles: 1000, TLB Size: 32, TLB Latency: 2
Block Size: 64, V2B Block Offset: 2, Memory Latency: 30
Number of Requests: 46
Trace file path is null.
Starting simulation...
Simulation started.
Simulation reset
```

Beispiel einer Tracefile mit den korrekten Werten der erforderlichen Ausgaben, die wir erhalten haben



```

unsigned calculate_primitive_gates(unsigned tlb_size, unsigned block_size, unsigned v2b_block_offset,
    unsigned base_gates = 1000; // Gates required for basic circuitry

    // Calculate gates for storing TLB entries
    unsigned bits_per_entry = 32 * 2 + 1; // tag (32 bits), physicalAddr (32 bits), valid (1 bit)
    unsigned storage_gates_per_entry = bits_per_entry * 4; // 4 gates per bit for storage
    unsigned total_storage_gates = tlb_size * storage_gates_per_entry;

    // Adding gates for data path logic
    // Assuming each addition of two 32-bit numbers requires approximately 150 gates
    unsigned datapath_gates = tlb_size * 150; // Assuming that in every TLB entry we will require one

    // Combining all gates
    unsigned total_gates = base_gates + total_storage_gates + datapath_gates;

    return total_gates;
}

```

die Formel zur Berechnung der Gatecount, die wir mit Hilfe der Online-Dokumentation für richtig hielten

Die TLB-Simulation implementiert die Adressübersetzung mit ausreichender Genauigkeit, wobei die Korrektheit durch automatische Tests und repräsentative Beispiele validiert wird. Die Verwendung von SystemC gewährleistet eine präzise Modellierung, während die Methoden `tlb_lookup` und `tlb_update` TLB-Einträge effektiv verwalten. Die Ergebnisse umfassen detaillierte Hit/Miss-Zahlen und Cycle-metriken, wie bereits erwähnt.

# Schaltkreisanalyse

- TLB Struktur:
- TLB besteht aus mehreren Einträgen, jeder Eintrag hat einen Tag, eine physische Adresse und ein Gültigkeitsbit.
- Jeder Speicherzugriff führt zu einer Überprüfung im TLB, ob die Adresse bereits vorhanden ist (Hit) oder nicht (Miss).

Grobe Schätzung der primitiven Gatter:

- Berechnungsgrundlage(Annahme):
  - Jeder TLB-Eintrag benötigt 32 Bits für den Tag, 32 Bits für die physische Adresse und 1 Bit für die Gültigkeit.
  - Für die Speicherung eines Bits werden etwa 4 primitive Gatter benötigt.
  - Addition von zwei 32-Bit Zahlen benötigt etwa 150 primitive Gatter.
- Ergebnisse:
  - TLB-Größe 32:
    - $32 \text{ Einträge} \times (65 \text{ Bits} \times 4 \text{ Gatter/Bit} + 150 \text{ Gatter}) = 8960 \text{ Gatter}$
  - TLB-Größe 64:
    - $64 \text{ Einträge} \times (65 \text{ Bits} \times 4 \text{ Gatter/Bit} + 150 \text{ Gatter}) = 17920 \text{ Gatter}$
  - TLB-Größe 128:
    - $128 \text{ Einträge} \times (65 \text{ Bits} \times 4 \text{ Gatter/Bit} + 150 \text{ Gatter}) = 35840 \text{ Gatter}$
  - TLB-Größe 256:
    - $256 \text{ Einträge} \times (65 \text{ Bits} \times 4 \text{ Gatter/Bit} + 150 \text{ Gatter}) = 71680 \text{ Gatter}$
  - TLB-Größe 512:
    - $512 \text{ Einträge} \times (65 \text{ Bits} \times 4 \text{ Gatter/Bit} + 150 \text{ Gatter}) = 143360 \text{ Gatter}$



# Vergleich bei Skalierung des Schaltkreises

Hit-Rate und Miss-Rate:

- Größere TLBs haben eine höhere Hitquote und eine niedrigere Missquote.
- Dies führt zu weniger Speicherzugriffen und einer besseren Gesamtleistung.

Zyklusanalyse:

- Mit zunehmender TLB-Größe reduziert sich die Anzahl der benötigten Zyklen, um alle Speicherzugriffe zu bearbeiten.
- Dies ist auf die reduzierte Anzahl der Misses und die effizientere Adressübersetzung zurückzuführen.





# Bewertung der Ergebnisse

## Effizienzsteigerung:

- Die Ergebnisse zeigen, dass eine größere TLB-Größe zu einer signifikanten Verbesserung der Speicherzugriffseffizienz führt.
- Die Anzahl der primitiven Gatter steigt jedoch linear mit der TLB-Größe an, was höhere Hardwarekosten bedeutet.

## Ursachen für Erkenntnisse:

- Die verbesserten Trefferquoten bei größeren TLBs sind auf die höhere Wahrscheinlichkeit zurückzuführen, dass eine Adresse bereits im TLB gespeichert ist.
- Die reduzierten Fehlertreffer führen zu weniger Hauptspeicherzugriffen und somit zu einer verbesserten Leistung.

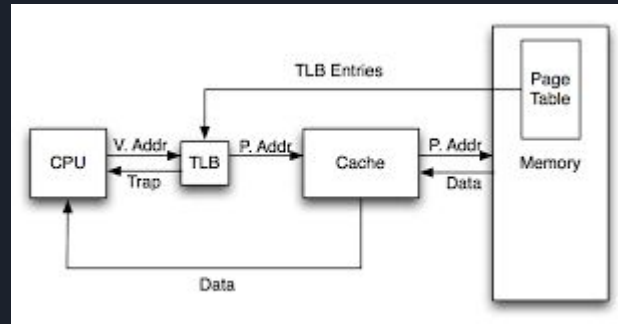


## Performance Comparison

TLB-Größe	Hitquote	Missquote	Zyklen
32	63%	37%	870
64	73%	27%	770
128	84%	16%	660
256	90%	10%	600

# Zusammenfassung und Ausblick

Insgesamt hat unser Projekt die Notwendigkeit einer TLB in modernen Prozessoren gezeigt, indem wir die Geschwindigkeiten von Operationen mit und ohne ihre Verwendung analysiert haben. Wir haben Vektoren als Datentyp verwendet, um die TLB zu repräsentieren, und dabei System C verwendet, indem wir Module, Gleichzeitigkeitsmanagement und Signalkommunikation definiert haben. Rückblickend hätte eine detailliertere Betrachtung der Speicherkohärenz und der Speicherhierarchie zu einer höheren Genauigkeit führen können. Optimierungen in der Simulationslogik und den Datenstrukturen wären ebenfalls möglich gewesen, z.B. die Verwendung eines Arrays als Datenstruktur zur Darstellung einer TLB oder die Minimierung des Speicherplatzes bei der Schleife.





# Quelle

- [https://en.wikipedia.org/wiki/Translation\\_lookaside\\_buffer](https://en.wikipedia.org/wiki/Translation_lookaside_buffer)
- <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- <https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf>
- <https://developer.arm.com/documentation/ddi0487/ka>
- <https://arstechnica.com/>
- <https://www.anandtech.com/>