LEHRSTUHL FÜR DESIGN AUTOMATION

Grundlagenpraktikum: Rechnerarchitektur

Cache Simulation und Analyse (A12)

Projektaufgabe – Aufgabenbereich Cacheassoziativität

1 Organisatorisches

Auf den folgenden Seiten finden Sie die Aufgabenstellung zu Ihrer Projektaufgabe für das Praktikum. Die Rahmenbedingungen für die Bearbeitung werden in der Praktikumsordnung festgesetzt, die Sie über Artemis[1] aufrufen können.

Anders als in den Hausaufgaben sind in diesem Projekt keine Module vorgegeben. Besprechen Sie innerhalb Ihrer Gruppe, welches Abstraktionsniveau für die Implementierung sinnvoll ist und diskutieren und konkretisieren Sie den Entwurf der Module gemeinsam. Die Teile der Aufgabe, in denen C-Code anzufertigen ist, sind in C nach dem C17-Standard zu schreiben. Die Teile der Aufgabe, in denen C++-Code anzufertigen ist, sind in C++ nach dem C++14-Standard zu schreiben. Die jeweiligen Standardbibliotheken sind Teil der Sprachspezifikation und duerfen ebenfalls verwendet werden. Als SystemC-Version ist SystemC 2.3.3 zu verwenden.

Der **Abgabetermin** ist **Sonntag 21. Juli 2024, 23:59 Uhr (CET)**. Die Abgabe erfolgt per Git in das für Ihre Gruppe eingerichtete Projektrepository auf Artemis.

Die **Abschlusspräsentationen** finden in der Zeit vom **19.08.2024 bis 30.08.2024** statt. Weitere Informationen werden noch bekannt gegeben. Beachten Sie, dass die Folien für die Präsentation am obigen Abgabetermin im PDF-Format abzugeben sind und keine nachträglichen Änderungen akzeptiert werden können.

Bei Fragen/Unklarheiten in Bezug auf den Ablauf und die Aufgabenstellung wenden Sie sich bitte **schriftlich** über Zulip an Ihren Tutor.

Wir wünschen Ihnen viel Erfolg und Freude bei der Bearbeitung Ihrer Aufgabe!

Mit freundlichen Grüßen
Die Praktikumsleitung

[1] `https://artemis.ase.in.tum.de`

2 Cache Simulation und Analyse

In modernen performanten Computersystemen spielt die Geschwindigkeit von Speicher und Prozessoren eine entscheidende Rolle für die Gesamtleistung. Ein zentrales Problem hierbei ist das Missverhältnis zwischen der Geschwindigkeit von Speicherzugriffen und der Rechengeschwindigkeit der Prozessoren. Dieses Problem wird häufig als *Von-Neumann-Flaschenhals* bezeichnet. Der Von-Neumann-Flaschenhals beschreibt die Limitierung der Systemleistung durch die begrenzte Bandbreite und die relativ langsamen Datenübertragungsraten zwischen Prozessor und Hauptspeicher.

Cache-Speicher

Um den Von-Neumann-Flaschenhals zu mildern, setzen moderne Computersysteme auf verschiedene Strategien. Dazu zählen die Verwendung von kleineren, schnelleren Zwischenspeichern: *Caches*. Caches sind spezielle Speicherbereiche, die dazu dienen, häufig benötigte Daten und Instruktionen vorzuhalten und somit die Zugriffszeiten erheblich zu verkürzen. Sie wirken als Puffer zwischen dem schnellen Prozessor und dem vergleichsweise langsamen Hauptspeicher.
Die Anwendungsmöglichkeiten von Caches gehen über das zwischenspeichern von Daten in einem Programm hinaus. Weitere Einsatzgebiete sind unter anderem:

- **Instruktions-Cache:** Auch das Programm selbst gehört zu den Daten die für das Ausführen einer Operation wichtig sind. Schließlich müssen nicht nur die Operanten, sondern auch die Operationen selbst aus dem Speicher geladen werden. Daher ist es sinnvoll, auch Instruktionen zu cachen.

- **Translation Lookaside Buffer (TLB):** Moderne Mehrbenutzersysteme verwenden virtuellen Speicher, um unter anderem Programme voneinander zu isolieren. Dabei werden (virtuelle) Speicheradressen von Anwendungen auf physische Speicheradressen abgebildet. Damit diese Abstraktion für das Programm transparent bleibt, muss das System bei jedem Speicherzugriff die virtuelle Adresse in eine physische Adresse übersetzen. Die Übersetzungstabelle liegt jedoch selbst im Hauptspeicher und Zugriffe benötigen viele Zyklen. Hier kommt der *Translation Lookaside Buffer (TLB)* zum Einsatz, ein spezieller Cache, der die Adressübersetzung beschleunigt. Die TLB speichert die Ergebnisse früherer Adressübersetzungen, sodass häufig genutzte Speicheradressen schneller zugänglich sind.

Aufbau von Caches

Der Aufbau von Caches spielt eine zentrale Rolle für deren Leistungsfähigkeit. Wichtige Aspekte hierbei sind Latenz, Assoziativität, Cachezeilen und die Größe der Cachezeilen.

- **Latenz:** Die Latenz bezieht sich auf die Verzögerung, die auftritt, wenn Daten aus dem Cache abgerufen werden. Je geringer die Latenz, desto schneller können die benötigten Daten bereitgestellt werden, was die Gesamtleistung des Systems

verbessert. Cache-Latenzen hängen unter anderem von der Assoziativität des Caches und der Größe des Caches ab. Es wird zwischen der Latenz bei einem Treffer(Hit-Latency) und der Latenz bei einem Miss(Miss-latency) unterschieden.

- **Assoziativität:** Assoziativität beschreibt, wie flexibel ein Cache Daten speichern kann. Ein vollständig assoziativer Cache kann Daten an jeder Stelle speichern, während ein direkt abgebildeter Cache Daten nur an einer bestimmten Stelle speichern kann. Höhere Assoziativität kann die Cache-Trefferquote verbessern, erhöht jedoch auch die Komplexität und möglicherweise die Zugriffszeit.
- **Cachezeilen:** Der Cache ist in mehrere kleine Blöcke unterteilt, die sogenannten Cachezeilen. Jede Cachezeile speichert einen bestimmten Datenblock aus dem Hauptspeicher. Wenn der Prozessor auf Daten zugreift, überprüft er, ob diese Daten in einer der Cachezeilen vorhanden sind.
- **Zeilengröße:** Die Größe der Cachezeilen ist ebenfalls entscheidend. Größere Cachezeilen können mehr Daten auf einmal speichern, was bei sequentiellen Speicherzugriffen von Vorteil sein kann. Allerdings kann dies auch zu mehr unnötigen Datenzugriffen im Hauptspeicher führen, wenn der Prozessor häufig auf nicht aufeinanderfolgende Speicheradressen zugreift.

3 Aufgabenstellungen

Im Rahmen Ihrer Aufgabe werden Sie die unterschiede Zwischen einem direkten und einem voll assoziativen Cache untersuchen. Ihre Aufgaben für das SystemC Systemdesign Finalprojekt lassen sich in die Bereiche Recherche und Implementierung (praktisch) aufteilen. Nutzen Sie die Ergebnisse Ihrer Recherche, um im Vortrag von der Relevanz des Themas zu überzeugen und die Korrektheit der Implementierung zu untermauern. Geben Sie einen kurzen Überblick der Rechercheergebnisse auch im Readme-File an. Die Antworten auf die Implementierungsaufgaben werden durch Ihren Code reflektiert. Die Implementierung stellt gemeinsam mit dem Vortrag den Hauptteil des Projekts dar. Nutzen Sie Abstraktion sinnvoll, um das System — (Speicher Cache - CPU) — zu simulieren. Dokumieren Sie in der Datei `Readme.md` den persönlichen Beitrag jedes Gruppenmitglieds.

3.1 Theoretischer Teil

Recherchieren Sie übliche Größen für direkt und voll assoziativen Cache sowie Hauptspeicher- und Cachelatenzen in modernen Prozessoren. Untersuchen Sie das Speicherzugriffsverhalten eines speicherintensiven Algorithmus. Erstellen Sie *csv*-Dateien, die beispielhaft die Speicherzugriffe des Algorithmus beschreiben. Beobachten Sie mithilfe Ihrer Implementierung das Verhalten der Cache-Architekturen in Bezug auf die Zugriffszeiten. Verwenden Sie die *csv*-Dateien, um die Simulation zu testen und diskutieren Sie *kurz* die Ergebnisse. Dokumentieren Sie ihre Ergebnisse in der Datei `Readme.md`. Stellen Sie mindestens ein Fallbeispiel fuer das Speicherzugriffsmuster Ihres ausgewaelten Algorithmus bereit.

3.2 Praktischer Teil

Um die Leistungsfähigkeit von Caches zu untersuchen, sollen Sie eine Cache-Simulation in SystemC mit C++ und ein Rahmenprogramm in C implementieren. Der Simulation werden Parameter übergeben, die die Größe und das Verhalten des Caches beschreiben, sowie eine Liste von Speicherzugriffen. Entwerfen Sie eine geeignete Struktur/Modul-Architektur für Ihre Implementierung.

Rahmenprogramm

Das Rahmenprogramm ist in C zu implementieren. Ihr Rahmenprogramm muss bei einem Aufruf die folgenden Optionen entgegennehmen und verarbeiten können. Ihr Rahmenprogramm soll selbe Eigenschaften bei der Verarbeitung von Kommandozeilenargumenten vorweisen wie Linux Kommandozeilenprogramme. Beispielsweise sind Reihenfolge und Leerzeichen zwischen Optionen und Argumenten großteils irrelevant. Wenn möglich soll das Programm sinnvolle Standardwerte definieren, sodass nicht immer alle Optionen gesetzt werden müssen. Ihr Rahmenprogramm soll fehlerhafte oder im Kontext ihrer Implementierung nicht sinnvolle Optionen korrekt abfangen und

mit einer aussagekräftigen Fehlermeldung auf `stderr` und einer kurzen Erläuterung zur Benutzung terminieren. Wir empfehlen Ihnen, die Kommandozeilenparameter mittels `getopt_long`[2] zu parsen. `getopt_long` akzeptiert auch Optionen, die nicht zur Gaenze ausgeschrieben sind. Ob Sie dieses Verhalten fuer Ihr Programm verwenden moechten, bleibt Ihnen ueberlassen.

- `-c <Zahl>/--cycles <Zahl>` — Die Anzahl der Zyklen, die simuliert werden sollen.
- `--directmapped` — Simuliert einen direkt ~~assoziativen~~ abbildenden Cache.
- `--fullassociative` — Simuliert einen voll assoziativen Cache.
- `--cacheline-size <Zahl>` — Die Größe einer Cachezeile in Byte.
- `--cachelines <Zahl>` — Die Anzahl der Cachezeilen.
- `--cache-latency <Zahl>` — Die Latenzzeit eines Caches in Zyklen. Latenzzeit ist unabhängig von Treffer oder Miss, sowie von Lese oder Schreiboperationen.
- `--memory-latency <Zahl>` — Die Latenzzeit des Hauptspeichers in Zyklen. Lese und schreiboperationen haben die gleiche Latenz.
- `--tf=<Dateiname>` — Ausgabedatei für ein Tracefile mit allen Signalen. Wenn diese Option nicht gesetzt wird, soll kein Tracefile erstellt werden.
- `<Dateiname>` — Positional Argument: Die Eingabedatei, die die zu verarbeitenden Daten enthält.
- `-h/--help` — Eine Beschreibung aller Optionen des Programms und Verwendungsbeispiele werden ausgegeben und das Programm danach beendet.

Sie dürfen weitere Optionen implementieren, beispielsweise um vordefinierte Testfälle zu verwenden. Ihr Programm muss jedoch nur unter Verwendung der oben genannten Optionen verwendbar sein.

Eingabedatei

Die Eingabedatei ist im *csv*-Format. Jede Zeile stellt eine Anfrage dar. Die erste Spalte gibt an, ob es sich um einen Lese- oder Schreibzugriff handelt. Die zweite Spalte gibt die Adresse an, auf die zugegriffen wird. Die dritte Spalte gibt den Wert an, der geschrieben wird, falls es sich um einen Schreibzugriff handelt. Die Adresse und der Wert sind in dezimaler oder hexadezimaler Darstellung angegeben. Bei einem Lesezugriff muss der Wert in der dritten Spalte leer sein. Ihre Implementierung soll beim Einlesen von Dateien die nicht diesem Format entsprechen, mit einer aussagekräftigen Fehlermeldung auf `stderr` und einer kurzen Erläuterung zur Benutzung terminieren.

[2] Siehe `man 3 getopt_long` für Details

W	0xabc	1337
W	0	1
R	1	
R	0x0	
W	0	0x1337
⋮	⋮	⋮

Tabelle 1: Beispiel einer Eingabedatei (tabellarisch dargestellt)

3.3 Simulation

Implementieren Sie in C++ die Funktion:

```
struct Result run_simulation(
  int cycles,
  int directMapped,
  unsigned cacheLines,
  unsigned cacheLineSize,
  unsigned cacheLatency,
  unsigned memoryLatency,
  size_t numRequests,
  struct Request requests[numRequests],
  const char* tracefile);
```

Der parameter `cycles` gibt die Anzahl der Zyklen an, die simuliert werden sollen. Der Parameter `directMapped` gibt an, ob ein direkt assoziativer abbildender Cache simuliert werden soll. Ein Wert von 0 simuliert einen voll assoziativen Cache, alle anderen Werte simulieren einen direkt assoziativen abbildenden Cache. Die Parameter `cacheLines` und `cacheLineSize` geben die Anzahl der Cachezeilen und die Größe einer Cachezeile in Byte an. Ihr Programm muss nur Zweierpotenzen als Cachezeilengroessen unterstuetzen. Die Parameter `memoryLatency` und `cacheLatency` geben die Latenzzeit des Hauptspeichers und des Caches in Zyklen an. Der Cache soll `write-through` sein. `write-through` bedeutet, dass Schreibzugriffe direkt im Hauptspeicher aktualisiert werden. Das ist in diesem Fall notwendig, da das einfache Eingabeformat keine Operationen unterstuetzt, die das Schreiben in den Hauptspeicher erzwingen. Solche Operationen sind in der Praxis notwendig, um zum Beispiel `Memory-Mapped I/O` zu implementieren. Die Anzahl der Anfragen ist in `numRequests` gespeichert. Die Anfragen sind in dem Array `requests` gespeichert. Der Parameter `tracefile` gibt den Dateinamen an, in dem das Tracefile gespeichert werden soll. Wenn `tracefile` `NULL` ist, soll kein Tracefile erstellt werden. Verwenden Sie LRU als Ersatzstrategie. LRU steht für *Least Recently Used* und ersetzt diejenige Cachezeile, die am längsten nicht benutzt wurde.

Die Funktion führt eine Simulation des Systems mit den gegebenen Argumenten durch und gibt eine `struct` mit den Ergebnissen zurück. Dabei sollen die Anfragen in der Reihenfolge bearbeitet werden, in der sie in dem Array `requests` übergeben werden. Eine

neue Anfrage darf erst bearbeitet werden, wenn die vorherige Anfrage abgeschlossen ist. Bei Lesezugriffen soll der gelesene Wert in das `data`-Feld der Anfrage geschrieben werden. Die `structs` `Request` und `Result` sind wie folgt definiert:

```
struct Request {
  uint32_t addr;
  uint32_t data;
  int we;
};
```

Das Feld `we` gibt an, ob es sich um einen Lese- oder Schreibzugriff handelt (0 für Lesezugriff, 1 für Schreibzugriff). Das Feld `data` enthält den Wert, der geschrieben werden soll, falls es sich um einen Schreibzugriff handelt. Das Feld `addr` enthält die Adresse, auf die zugegriffen wird. Der Datenbus und der Adressbus sind, wie aus der `struct` zu entnehmen, 32 Bit breit. Der Speicher ist byte-adressiebar.

```
struct Result {
  size_t cycles;
  size_t misses;
  size_t hits;
  size_t primitiveGateCount;
};
```

Die Funktion gibt ein `struct` mit den Ergebnissen der Simulation zurück. Das `cycles`-Feld soll die Anzahl der Zyklen angeben, die benötigt wurden um alle Anfragen abzuarbeiten. Wenn nicht alle Anfragen innerhalb der gegebenen Zyklen bearbeitet werden konnten, soll `cycles` den Wert `SIZE_MAX` haben. Die Felder `misses` und `hits` sollen jeweils die Anzahl der Misses und Hits angeben. Das Feld `primitiveGateCount` ist eine Schätzung der Anzahl der Gatter, die benötigt werden, um den das System in Hardware zu implementieren. Diese Schätzung soll nur Gatter zählen, die direkt für den Beschleuniger benötigt werden. Dazu gehören die Gatter, die für die Datenspeicherung und Ersetzungsstrategie notwendig sind; Hauptspeicher und Simulationslogik sollen nicht gezählt werden. Die Abschätzung muss nicht exakt sein. Beispielweise genügt es für das Speichern von einem Bit 4 und für eine Addition von zwei 32-Bit Zahlen ≈ 150 primitive Gatter zu veranschlagen.

Sie dürfen innerhalb der Funktion davon ausgehen, dass die übergebenen Request-*Structs* valide sind. Alle überprüfungen von fehlerhaften Eingaben sollen bereits in dem Rahmenprogramm durchgeführt werden. Stellen Sie sicher, Sie diese Funktion auch von C Programmcode aufrufbar ist[3].

[3] Siehe `https://en.cppreference.com/w/cpp/language/language_linkage`

4 Überblick über abzugebende Dateien

- `Readme.md` - Dokumentation der Ergebnisse des theoretischen Teils und ro Gruppenmitglied ein kurzer Abschnitt über den persönlicher Beitrag
- `Makefile` - Makefile für die Kompilierung *einer* ausführbaren Datei
- `src/` - Verzeichnis für den Quellcode
  - C-Code für das Rahmenprogramm
  - C++-Code für die Cache-Simulation in SystemC
- `examples/` - Verzeichnis fuer *csv*-Fallbeispiele
- `slides/` - Verzeichnis für die Präsentationsfolien und weitere Materialien für den Vortrag. Während des Vortrags werden allerdings nur die Datei `slides.pdf` auf dem Beamer präsentiert.
  - `slides.pdf` - Präsentationsfolien im PDF-Format. Während des Vortrags wird dieses Dokument präsentiert.

Sie dürfen weitere Dateien und Verzeichnisse erstellen, wenn Sie diese für Ihre Implementierung benötigen.