Praktikum Rechnerarchitektur SS 2024

Aufgabe 4a und 4b

Ausgabedatum: 20.05.2024 (4a) &
3.6.2024 (4b)
v 2.0

Cache Simulation

In diesem Praktikum geht es darum, Ihr Verständnis von Caches zu festigen Dazu schreiben Sie ein Java Programm[1], welches das Verhalten eines Caches beliebiger Größe simuliert. In dieser Aufgabe befassen wir uns zunächst (4a) mit einem **direkt abgebildeten Cache** (direct-mapped cache), in Praktikumsaufgabe 4b erweitern Sie Ihre Lösung, so dass auch assoziative Caches simuliert werden können.

Ihr Cache Simulator executable sollte folgende Kommandozeilenargumente akzeptieren:
`-s <s>` Anzahl der Indexbits s, S = 2^s ist die Anzahl der Cache Blöcke (bzw. Sätze beim assoziativen Cache)
`-E <E>` Assoziativität des Caches, E = Anzahl der Blöcke pro Satz diese Option müssen Sie erst in Praktikum 5 implementieren, d.h. momentan ist hier nur E = 1 zugelassen
`-b <b>` Anzahl der Block Bits, B = 2^b ist die Blockgröße (in Byte)
`-t` <tracefile> der Name der valgrind Trace Datei, die Ihr Simulator simulieren soll
`-v` aktiviert den „verbose“ Mode, bei dem Ihr Cachesimulator für jede eingelesene Trace Zeile das aktuelle Cache Verhalten ausgibt (siehe unten).
`-h` gibt die verfügbaren Optionen Ihres Simulators aus, d.h. die Beschreibung dieser Flags.

Wichtig: benennen Sie diese Optionen nicht um, Sie können weitere Optionen hinzufügen, aber obige Optionen müssen als Minimum unterstützt werden. *Wenn Sie die Java Vorlage verwenden, ist das Parsen dieser Kommandozeilenoptionen bereits für Sie implementiert.*

Valgrind Trace Dateien

Valgrind ist ein (Linux) Werkzeug, mit dem man die Hauptspeicherzugriffe eines Programmes mitprotokollieren kann.

```
valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

schreibt zum Beispiel die Speicherzugriffe beim Ausführen des Befehls „ls – l“ auf die Standardausgabe.

Valgrind Speichertraces haben 4 verschiedene Trace Einträge: I für Instruction Cache Zugriffe, S für Store Zugriffe, L für Daten Load Zugriffe und M für Modify, d.h. kombinierte Load und Store Zugriffe. Vor dem I steht nie ein Lehrzeichen, vor M, L oder S steht immer eines. Nach der Art des Speicherzugriffes folgt ein Leerzeichen und nach die 64bit Adresse

[1] Sie können auch C / C++ oder eine andere Sprache verwenden, aber dafür gibt es keine Vorlage.

auf die Zugegriffen wurde. Dieser folgt (getrennt durch ein Komma) die Größe des Speicherzugriffs in Byte.

Im Folgenden sehen Sie ein Beispiel für jede Art von Speicherzugriff.

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Ihr Cache Simulator bekommt in dem Argument `-t <tracefile>` einen Speichertrace dieser Form, also eine Folge von Speichergriffen, so wie sie von valgrind protokolliert werden. Zeilen, die nicht diesem Schema folgen, soll Ihr Simulator ignorieren.

Diese Folge von Speicherzugriffen muss Ihr Simulator dann verwenden, um das mit den anderen Parametern spezifiziertes Cache zu simulieren. **Wenn Sie ein assoziatives Cache simulieren, implementieren Sie einen perfekten LRU Ersetzungsmechanismus** (für Praktikum 4a (=4) müssen Sie allerdings nur einen direkt-abgebildeten Cache simulieren).

Das Parsen der valgrind Dateien ist für Sie in der Vorlage bereits implementiert.

Simulator Ausgabe

Für Ihren Simulator Sie I-Cache Verhalten ignorieren, sie müssen also nur den Daten Cache simulieren. Am Ende des Simulation soll Ihr Simulator eine Zusammenfassung der Simulation (falls nicht im –v Modus) ausgeben:

```
java -jar simulator.jar  -s 4 -E 1 -b 6 -t test_trace.txt
```

Ausgabe:

```
Running cache simulation with trace file test_trace.txt
Cache lines: 16
Block size in bytes: 64
Associativity: 1
Simulation starting...
offset bits: 6 indexBits 4
Successfully simulated 61 valgrind lines
Access sizes in trace:
1 2 3 4 5 6 7 8 10
hits: 6 misses: 8 evictions: 1
Simulation finished.
```

Dabei ist jede Verdrängung [2](eviction) auch ein Fehlzugriff (miss), d.h. in der obigen Statistik gibt es 8 misses von denen 1 zu einer eviction geführt hat.

[2] Unter einer Verdrängung (engl. Eviction“) versteht man ein Fehlzugriff mit Konflikt („conflict miss“) bei dem ein bereits existierender gültiger Cache Eintrag durch einen neuen ersetzt wird.

Ausgabe im Verbose Modus

Im –v Modus muss Ihr Simulator außer der bereits beschriebenen Ausgabe für jeden (Daten) Speicherzugriff ausgeben, ob er ein hit, miss oder eviction[3] darstellt. Dazu soll zusätzlich die entsprechende Zeile in der Trace Datei angezeigt werden, deren Speicherzugriff gerade simuliert wird. Sie können auch noch, wie bei der Ausgabe unten, zusätzliche nützliche Infos mitprotokollieren, z.B. das das tag nicht übereinstimmte.

Am Ende geben Sie auch noch den ganzen Inhalt des Caches (d.h. die Tags) aus; das ist für Debugging sehr nützlich.

Diese Ausgabe sieht bei mir so aus und sollte bei Ihnen ähnlich aussehen:

java -jar simulator.jar -v -s 4 -E 1 -b 6 -t test_trace.txt

Running cache simulation with trace file test_trace.txt

Cache lines: 16

Block size in bytes: 64

Associativity: 1

Simulation starting...

offset bits: 6 indexBits 4

store ffeffffb8 8 miss

store ffeffffb0 8 hit

store ffeffffa8 8 hit

store ffeffffa0 8 hit

store ffeffff98 8 hit

store ffeffff90 8 hit

store ffeffff88 8 hit

load 4222e70 8 miss

store 4222c98 8 miss

load 4223000 8 miss

updating index 0 with tag: 67724

store 4223a08 8 miss

store 42239f8 8 miss

store 4223aa8 8 miss

load 4222e80 8 miss

eviction

Successfully simulated 61 valgrind lines

Access sizes in trace:

1 2 3 4 5 6 7 8 10

hits: 6 misses: 8 evictions: 1

3

Dumping cache contents:

index: 0: 67724

index: 1:

index: 2: 67723

index: 3:

index: 4:

index: 5:

index: 6:

index: 7: 67726

index: 8: 67726

index: 9: 67723

index: 10: 67723

index: 11:

index: 12:

index: 13:

index: 14: 67092479

index: 15:

Simulation finished.

Jede Zeile gibt dabei die Art des Speicherzugriffs an (L, M, S, wie oben definiert), dann die Zeile in der Trace Datei aus welcher der Speicherzugriff stammt, und schließlich 1 miss für einen Fehlzugriff, 1 hit für einen Treffer und bei mehreren Zugriffen (M) in derselben Zeile 1 hit miss. Zusätzlich muss wenn eine Verdrängung („eviction“) statt findet, nach dem miss ein „eviction“ stehen.

Implementierung und Abgabe

Sie können zur Implementierung die im Handout enthaltenen Dateien verwenden. Alternativ können Sie diese auch in einem Docker Container in denen Ihnen alle notwendige Software, Trace Dateien und ein Test-Script als ein Ubuntu-System bereit steht.

Der Containter ist in der public Docker Registry als `markusmock/ib345:simulator` verfügbar:

```
docker run -it markusmock/ib345:simulator bash
cd simulator/src/
make
```

und Sie können dann in CacheSimulator,java Ihren Simulator entwickeln. In dem Container steht auch schon git bereit, so dass Sie Ihre Lösung bequem während der Entwicklung sichern können (denken Sie daran, dass jede Veränderung im Container, wenn Sie diesen nicht speichern, verloren sind!).

Für Ihre Java Implementierung stehen Ihnen folgende Dateien bereit:

```
CacheSimulator.java
Main.java
```

```
ValgrindLineParser.java
commons-cli-1.4.jar
```

Verwenden Sie eine IDE (Eclipse, IntellliJ oder Visual Studio Code empfohlen!) müssen außerdem die jar-Datei mit in Ihr Java Projekt auf den classpath nehmen, da der ValgrindLineParser Funktionalität daraus verwendet.

Auf der Kommandozeile kompilieren Sie z.B. so:

```
javac -cp commons-cli-1.4.jar *.java CacheSimulator.java Main.java
ValgrindLineParser.java
```

Die Abgabe dieses Aufgabe besteht aus zwei Teilen: (1) Vorzeigen und Erklären Ihrer Lösung im Praktikum und (2) Hochladen Ihres Java Codes in Moodle.

Implementierungstipps

Cache Simulatoren sind ein in der Praxis unverzichtbares Werkzeug beim Entwurf neuer Computerarchitekturen und des Speichersystems für diese. Sie müssen daher sehr effizient implementiert werden, damit das Verhalten von realistischen Programmen in vertretbarer Zeit simuliert werden kann. Ihr Simulator muss nicht diesen hohen Anforderungen genügen, aber Sie sollten bestimmte Fehler vermeiden, da diese die Performance zu stark beeinträchtig bzw. zu viel Hauptspeicher benötigen (und der mitgelieferte Trace muss von Ihrem Cache Simulator korrekt und insbesondere ohne Absturz verarbeiten). Insbesondere also:

- Versuchen Sie nicht, den Trace als Ganzes in den Hauptspeicher einzulesen und dann zu verarbeiten, sondern lesen Sie zeilenweise und simulieren dann jeweils die Zeile, die Sie eingelesen haben.
- Repräsentieren Sie den Cache möglichst effizient, also am besten durch Integer-Werte, die Sie effizient auch bitweise durch die Java-Operationen & | usw. manipulieren können. Für das Cache-Verhalten ist der gelesene bzw. geschriebene Wert nicht relevant (und im Trace auch gar nicht mitprotokolliert) sondern nur die Adresse des Zugriffs. Insbesondere sollten Sie Strings zur Repräsentation vermeiden, da das zu einer sehr langsamen Implementierung führen wird. Am besten Sie definieren Sich einen passenden Datentyp (Java Klasse) für das Cache und Cache-Einträge.
- Sie werden im praktischen Cache Trace denn Fall feststellen, dass Cache Zugriffe stattfinden, die sehr groß sind und insbesondere größer als ein Cache Block sind. (Diese behandeln Sie am einfachsten dadurch, dass Sie sie (intern in Ihrer Simulation) in mehrere Zugriffe zerlegen. Um Ihre Aufgabe einfach zu halten, können Sie diese dann einfach so simulieren, d.h. aus einer Valgrind-Zeile werden u.U. mehrere simulierte Cache Zugriffe.

Wie wissen Sie, ob Ihr Simulationsergebnis korrekt ist?

Es ist nicht ganz einfach, sich zu überzeugen, dass Ihr Simulator korrekt ist. Damit Sie das tun können, ist die Ausgabe die für die Traces und Cache-Konfigurationen von meiner (über Jahre getesteten) Implementierung mit in der Aufgabe dabei.

Lesen Sie die Hinweise in `readme.txt` wie Sie Ihre Ergebnisse mit den erwarteten vergleichen können. Sollten Sie andere Ergebnisse bekommen, liegt es mit großer Wahrscheinlichkeit an Ihrem Simulator. *Jeder der jedoch einen bis dahin nicht bekannten Fehler in meiner Implementierung findet, bekommt von mir eine Tüte Gummibärchen*.