CSim

ein Cache Simulator

Leonhard Wiedmann

10. April 2019

Inhaltsverzeichnis

1 Einleitung

Als Cache wird ein Speicher bezeichnet, welcher Daten aus einem entfernet Speicher nahe dem Ort ablegt, wo die Daten benötigt werden, um bei wiederholten Zugriff die Zugriffszeit zu verkürzen. Dabei werden verschiedene verfahren verwendet um den Cache optimal zu nutzen. Es gibt bis zum heutigen Zeitpunkt kein Perfektes Verfahren für alle Programme, da der Speicherzugriff in den Programmen sehr stark varriiert.
Im Rahmen eines Uni Praktikums ist ein Programm zu analysierung von Applikationen und ihrem Cacheverhalten entstanden. Die Idee ist ein Programm zu entwickeln, welches das reale verhalten von Applikationen analysiert und aufgrund der Speicherzugriffe verschiedene Caches simuliert um daraus den optimalen Cache für die Applikation zu finden. Diese Analysie kann bei der Entwicklung von Software behilflich sein, um dem Programmierer zu zeigen wie gut Code den Cache nutzen kann, aber auch für Hardware Synthese, um für Recheneinheiten einen optimalen Cache zu synthetisieren.

2 Beschreibung

Bei machen Programmen ist es sehr einfach durch betrachten des Codes einfach zu beschreiben, was für ein Cache optimal für das Programm ist. Da aber moderne Kompiler versuchen den Code schneller auszuführen, kann es auch dazu führen, dass die Applikation nach dem Kompilieren in einer anderen Reihenfolge ausgeführt wird. Des weiteren ist die Analyse durch betrachten des Codes bei Komplexeren Programmen, die mit anderen Bibliotheken arbeiten nur noch für Profies, welche alle Bibliotheken auswendig kennen, möglich.
Um die analyse zu Automatisieren wurde dieses Programm entwickelt, mit dem Namen CSim. Es soll das realistische Verhalten einer Applikation analysieren und einen für die Applikation optimalen, Cache zurückgeben.
Unter optimal wird dabei verstanden, dass der Cache mit dem minimum an Hardware umgesetzt werden kann. Daher wird in der Simulation auf Prefetching verzichtet, weil dabei auch Daten vorgeladen werden, welche nicht benötigt werden, was den Cache unnötig groß machen würde.

2.1 Eingabe

Um eine Applikation zu analysieren muss sie neu kompiliert und ausgeführt werden um die Datensätzen der Speicherzugriffe zu erstellen. Des weiteren muss der Nutzer aber dafür sorgen, dass das Programm unter realistischen bedingungen kompiliert und simuliert wird. Es sollte also mit realistischen Eingaben ausgeführt werden.
Für die Eingabe der Parameter zur Ausführung muss keine bestimmte Order eingehalten werden.

2.1.1 Pfad

Falls die Applikation keine Flags oder Eingaben benötigt, benötigt CSim zum ausführen nur den relativen Pfad zur simulierenden Applikation. Wenn man zum Beispiel das Programm *bsp.c* simulieren möchte, welches sich im aktuellen Pfad befindet kann es mit dem Befehl *cSimbsp.c* simuliert werden.

2.1.2 Flags (-flag=)

Als Flags werden alle zusätzliche Flags bezeichnet, welche vom LLVM Kompiler benötigt werden. Diese können durch -flag= gesetzt werden. Standartmäig werden keine Flags beim Kompilieren gesetzt.

2.1.3 Links (-link-files=)

Um eine Applikation zu kompilieren werden vom Kompiler auch verlinkte Dateien benötigt. Diese können durch den Parameter -flink-files= gesetzt werden. Dabei wird relativ zum aktuellen Pfad gesucht.

2.1.4 Input (-input=)

Des weiteren werden für eine realistische Ausführung auch Eingabeparameter benötigt. Diese können mit dem Parameter -input= hinzugefügt werden. Genauso wie die bei den verlinkten Dateien wird auch hier relativ zum aktuellen Pfad gesucht.

2.1.5 Erwartungswert (-e=0.9)

Der Erwartungswert *expectation* ist die Anzahl der im Cache vorhandenen Daten (*cachehits*) geteilt durch die Anzahl der Ladezugriffe *loads*. $expectation = \frac{cachehits}{loads}$
Ein Erwartungswert von 1.0 (= 100%) wird also dann erzielt, wenn der Cache alle Daten aus dem Hauptspeicher bereits vor der Ausführung im Cache hat. Dieser Wert ist theoretisch nicht möglich, da kein Prefetching angewendet wird, kann aber aufgrund von Rundungsfehlern dennoch auftreten. Für einen optimalen Cache, wäre das also die Vorraussetzung, leider kann selbst eine annäherung an 1.0 für viele Programm nicht erreicht werden. Es kann dafür zwei gründe geben. Entweder die Applikation möchte Daten aus dem Hauptspeicher, welche durch aktuelle Ersetzungsstrategien nicht erkannt wird und verdrängt benötigte Daten um sie kurz danach wieder zu laden oder der Cache ist zu klein. CSim simuliert den Cache immer mit allen ihm zur verfügung stehenden Strategien (welche im Kapitel Ersetzungsstrategien 3.2.4 erklärt werden). Wodurch der Bestmögliche Erwartungswert erreicht werden soll. Er versucht aber auch den kleinstmöglichen Cache zu finden, woraus folgt, dass er den kleinstmöglichen Cache mit dem größten Erwartungswert finden möchte.
Wenn der Nutzer den Erwartungswert verringern will um die Suche zu beschleunigen, kann er mit dem Parameter -e= verändert werden. Als Standartwert ist 0.9 gesetzt, dadurch soll die Simulationszeit verkürzt aber dennoch gute Ergebnisse erreicht werden.

2.1.6 Granularität (granularity g=0")

Die Granularität gibt die möglichkeit den Cache für einzelne Variablen zu simulieren. Als Standartwert ist 0 gesetzt, wodurch alle Lade- und Schreibzugriffe aus einem Datenstrom analysiert werden. Wenn der Wert auf 1 gesetzt wird, werden die Datenströme von den einzelnen Variablen analysiert und evaluiert. Dadurch kann ein optimaler Cache für die einzelnen Variablen gefunden werden.

2.1.7 Zusätzliche Parameter

Des weiteren können mit dem Parameter für Debuging (-d=) weitere Informationen über das Laufzeitverhalten von CSim und der Analyse angezeigt werden. Zum aktuellen Zeitpunkt wird nur -d=1 unterstützt, welches die Ergebnisse des besten Caches für jede

Iteration zurückgibt.
Mit dem Parameter -oD=1 wird kein neuer Datenstrom erstellt, das Funktioniert aber nur, wenn das Programm zuvor einen Datenstrom erstellt hat und ihn in seinem Standartverzeichnis abgespeichert hat. Dieser Parameter kann sinnvoll sein, wenn man den Erwartungswert höher oder niedriger ansetzt, das Programm aber nicht erneut komplett ausführen möchte.
Die Informationen über die Parameter können auch angezeigt werden, wenn man keinen Wert oder help eingibt.

2.2 Ausgabe

Nach dem simulieren der Applikation, wird eine Ausgabe im Terminal angezeigt, welche uns den optimalen Cache beschreibt. Als grundlegende Größen für einen Cache wird in diesem Programm die Anzahl der Zeilen und die Größe der Zeilen angegeben.

2.2.1 Treffer

Als erstes wird beschrieben, wie gut der Cache im besten Fall ist. Dies wird durch die read-hits und write-hits angezeigt. Die write-hits sind die Anzahl der aus dem Cache geladenenen Daten im vergleich zur Gesamtzahl der geladenen Daten. Woraus folgt, dass der Cache uns Anzeigt wie effizent er genutz werden konnte. Äquivalent ist es mit der Anzahl der Schreibzugriffe.

2.2.2 Schreibkonflikte

Um eine optimale Schreibstrategie für den Cache zu implementieren ist es von Vorteil zu erkennen wie viele Schreibkonflikte aufgetreten sind. Ein Schreibkonflikt tritt dann auf, wenn auf einen Datensatz im Cache geschrieben wird und dieser dann aber verdrängt wird. Was dazu führt, dass der Datensatz spätestens dann in den Hauptspeicher übertragen werden muss. Wenn aber sehr oft auf einen Wert geschrieben wird bevor er verdrängt wird, kann es sinnvoller sein ihn nicht unnötigen oft in den Hauptspeicher zu schreiben.

2.2.3 Größe

Die Größe des Caches ergibt sich aus der Anzahl der Zeilen multipliziert mit der größe der Zeilen. Diese Werte werden durch *lines* und *size* angegeben. Die Anzahl der Zeilen und die größe der Zeilen muss eine Zahl sein, die durch $2^{\mathbb{N}}$ dargestellt werden kann.

2.2.4 associativity

Des weiteren gibt es verschiedene Assoziativitäten für einen Cache. Um die Suche zu vereinfachen wird nur eine Assoziativität von 1, 2, 4, 8 und 16 simuliert. Eine Assoziativität von 1 bedeutet, dass der Cache Vollassoziativ ist und je höher die Zahl der Assoziativität, um so weniger Hardware wird benötigt um den Cache zu implementieren.

2.2.5 Ersetzungsstrategien

Um zusätlich die Cachezugriffe besser zu verstehen, ist es hilfreich zu erkennen, welche Ersetzungsstrategie die Effizienteste ist. Im aktuellen Simulator sind First-In-First-Out (FIFO), Least Frequently Used (LFU) und Least Reasently Used (LRU) implementiert. In der Ausgabe wird unter "Replacement Policy"vorgeschlagen, welche Ersetzungsstrategie am effizientesten ist. Das vorgehen der Ersetzungsstrategien wird im Kapitel Ersetzungsstrategien 3.2.4 erklärt.

2.2.6 Beispiel

Um das ganze etwas anschaulicher zu erklären hier eine Beispielausgabe.

```
read Hits 78.6781
write hits 58.2236
write conflicts 41.7696
lines 1024
line size 1024
associativity 2
replacement policy LFU
```

Diese Applikation wurde mit den Standartwerten für Granularität (0) und Erwartungswert (0.8) ausgeführt. Was bedeutet, dass das Programm nicht mehr als 78.6781 % erfüllen kann.
Außerdem treten in über 41 % der Schreibzugriffe ein Konflikt auf. Das bedeutet, dass bei über 41 % der Schreibzugriffen der Platz, den sie im Cache belegen nach dem Schreiben auf den Wert von einem anderen Wert verdrängt wird.
Die Optimale Cache größe ist 1024 Zeilen mal 1024 Byte größe, was 1048576 Bytes ergibt. Das entspricht der Größe von 1 MiB.
Als optimale Assoziativität wird eine 2 Fache Satzassoziativität vorgeschlagen. Dies bedeutet, ein Cache in dem 2 Sätze zu jeweils 512 Lines vorhanden sind wäre optimal.
Um diesen Cache zu verwalten wird die Least Frequently Used Strategie empfohlen.

3 Technik

Der Cache Simulator wurde komplett in C++ geschrieben, was zum einen Objektorientertes Programmieren einfach macht, zum anderen wurde ein LLVM Pass zur Datensammlung geschrieben, welcher C++ erfordert.

Im Internet gibt es aktuell einige verschiedene Arten einen Cache zu simulieren und auch einige Vorlagen, wie man einen Cache Simulator selbst programmiert. Dieser Simulator baut auf den Cache Simulator von Levindo Neto (https://github.com/levindoneto/Cache-Simulator) auf. Der Simulator von Levindo Neto ist in C geschrieben und gut dokumentiert. Er beinhaltet einen einfachen Simulator für Datenströme. In diesem Simulator werden die Werte für den gewünschten Cache eingegeben und dann mit einer Datei als Datenstrom simuliert.
Um daraus einen Cache simulator für alle Programme zu machen haben wir ihn von C nach C++ portiert, einen Generator für Datenströme und eine Evaluierung programmiert.

3.1 Datenstrom

Als erstes wird deshalb das zu simulierende Programm mit LLVM kompiliert. Bei der Kompilierung werden zusätzliche Befehle vor Lade- und Schreibzugriffen eingefügt, welche die Werte für die Speicherzugriffe in eine Datei speichern. In der Datei, wird in jeder Zeile ein Zugriff mit der absoluten Adresse, Größe und Art des Zugriffs geschrieben. Der Wert "140725679285040 W4ßum Beispiel beschreibt einen Schreibzugriff (W = write, R = read) mit der Größe von 4 Byte auf den Wert in der Adresse 140725679285040.

Diese Datei wird im Anschluss in den Hauptspeicher geladen, da diese Werte für die Evaluierung wiederholt gelesen werden muss. Beim Laden in den Hauptspeicher wird dabei jede Zeile in ein Objekt umgewandelt. Jedes dieser Objekte hat drei Attribute. Ein long unsigned für die Adresse, einen char für die Art des Zugriffs und ein integer Wert für die Größe.

3.2 Simulation

Wenn nun der Datenstrom bereit liegt fängt die Applikation an zu simulieren. Die Simulation des Caches übernimmt die Strategie von Levindo Neto. Der Cache simulator erstellt mehrere Arrays. Jeder hat die größe, Anzahl der Zeilen mal Größe der Zeilen.

3.2.1 Cache Daten

Um die Daten im Cache zu simulieren wird ein Array von booleans erstellt. Jeder Datenwert in dem Array simuliert ein Byte. Solange der Wert in der Position im Array False ist, bedeutet es, dass kein Wert an dieser Position gespeichert wurde.

3.2.2 Cache Adresse

Auserdem erstellt der Cache noch zusätzlich einen Array mit den aus dem Cache oberen Teil der Adresse. Dieser Teil dient dazu den Cache ordnungsgemäß zu adressieren.

3.2.3 Vorgehen

Der Simulator berechnet mit dem ihm zur verfügung stehenden Datenstrom das Verhalten eines realen Caches. Das bedeutet, er berechnet entsprechend des Cache Design den oberen Teil der Adresse und versucht anschließend mit der abhängig von der Assoziativität einen freien Platz im Cache zu finden. Dabei zählt CSim alle Zugriffe auf den Speicher und die vom Cache vorgeladenen Zugriffe. Zusätlich wird die Zeit simuliert. Dafür wird eine Variable mit 0 initaliesiert und bei jedem Zugriff auf den Cache um 1 erhöht.

3.2.4 Ersetzungsstrategien

Wenn aber der Platz im Cache bereits belegt ist, wird mit einer Ersetzungsstrategie ein alter Datensatz verdrängt. CSim vergleicht dabei Least-Frequently-Used (LFU), Least-Recently-Used (LRU) und First-In-First-Out (FIFO).
Am einfachsten ist das Vorgehen von FIFO zu erklären. Dafür wird ein Array in der größe des Caches erstellt und jedesmal, wenn ein Datensatz neu in den Cache kommt, der Wert der Zeit variable darin gespeichert. Wenn nun ein Datensatz verdrängt werden soll, wird der Datensatz mit dem geringsten Wert verwendet.
Das vorgehen von LFU ist ähnlich zum vorgehen von FIFO, mit dem Unterschied, dass nicht nur wenn ein Datensatz neu in den Cache kommt die Zeit variable aktuallisiert wird, sondern bei jedem weiteren Zugriff auf den Wert im Cache.
Das ist anders im LRU verfahren. Bei diesem Verfahren wird ein Array erstellt in dem die Anzahl der Zugriffe gezählt wird und der Datensatz mit der geringsten Anzahl wird verdrängt.

3.3 Evaluierung

Die Evaluierung wird in zwei Teile aufgetrennt. In jedem dieser Teile wird in jedem Simulationsschritt mit allen Ersetzungsstrategien parallel simuliert und im Anschluss das beste Ergebniss daraus ausgewählt.
Zuerst wird die Größe des Caches mit der komplexesten Assoziativität ermittelt. Dadurch wird die minimale Größe des Cache gefunden. Diese Suche ist in zwei Schleifen aufgetrennt um die Komplexität für große Werte zu minimieren. Grundsätzlich funktioniert

die Suche aber nach folgendem Schema. Zuerst wird ein Cache mit der Anzahl der Zeilen ($lines$) und der Größe der Zeilen ($size$) von 2 simuliert wird $[i1 : size = 2, lines = 2]$. Danach wird ein neuer Cache simuliert indem $lines$ und $size$ um den Vorherigen Wert mal 2 erhöht wird $[i2 : size = 2 * 1x, lines = 2 * 1x]$. Das Ergebniss wird dann mit einem Cache verglichen, indem nur $size$ erhöht wird $[i3 : size = 2 * 1x, lines = 2]$. Wenn das Ergebniss von i2 schlechter oder genauso gut wie i3 ist, wird i3 verwendet und fortgesetzt. Wenn aber i2 besser ist, wird nur die Anzahl der Zeilen erhöht $[i2 : size = 2, lines = 2 * 1x]$ und das Ergebniss davon mit i2 verglichen. Dadurch kann der minimal beste Cache gefunden werden.
Das wird so lange iteriert bis ein Cache simulator den Erwartungswert übertrifft oder schlechter ist als in der Vorherigen Iteration. Um die Anzahl der Simulationen aber dennoch zu verringern wird zuerst in Schritten von 2^{10}, dann in Schritten von 2^5, 2^2 und zum Schluss 2^1 erhöht. Auf diese Weise kann die Laufzeit verringert werden aber dennoch ein genaues Ergebniss ermittlt werden.
Dieses Vorgehen lässt sich besser anhand des Pseudocode fig. 3.1 erkennen.

Nachdem der minimale Cache mit dem optimalen Ergebnis gefunden wurde, wird versucht die Satzassoziativität zu erhöhen. Um so höher die Satzassoziativität um so weniger Hardware wird benötigt um den Cache zu bauen. Dafür wird eine Schleife ausgeführt in der bei jeder Iteration die Satzassoziativität erhöht wird. Wenn das Ergebnis der höheren Satzassoziativität genauso gut wie der Vorherige oder besser als der Erwartungswert ist, wird weiter erhöht, bis zu einer maximalen Satzassoziativität von 16.

```
simulate(size, count){
  FIFO = Cache(size, lines, "FIFO")
  LRU = Cache(size, lines, "LRU")
  LFU = Cache(size, lines, "LFU")

  return best_result(FIFO, LRU, LFU)
}

size = 1
lines = 1

increment = 10

best = simulate(size, lines)
last

while(best.result < expectation){
  tmp = simulate(size + 2^increment, lines + 2^increment)
  incSize = simulate(size + 2^increment, lines)
  if(tmp.result > incSize.result){
    incLine = simulate(size, lines + 2^increment)
    if(tmp.result > incLine.result){
      size += 2^increment
      lines += 2^increment
      best = tmp
    }
    else{
      lines += 2^increment
      best = incLine
    }
  }
  else {
    size += 2^increment
    best = incSize
  }

  if(last.result >= best.result){
    if(increment == 1){
      break;
    }
    else{
      increment /= 2
    }
  }
}
```

Abbildung 3.1: minimalen Cache finden

```
sim(assoc){
  FIFO = Cache_assoc(best_size, best_lines, "FIFO", assoc)
  LRU = Cache_assoc(best_size, best_lines, "LRU", assoc)
  LFU = Cache_assoc(best_size, best_lines, "LFU", assoc)

  return best_result(FIFO, LRU, LFU)
}

assoc = 1
best = sim(assoc)
last = sim(assoc)
while( best > expectation || last.result == best.result ){
  last = best
  assoc += 1
  if(assoc > 16){
    break;
  }
  best = sim(assoc)
}
```

Abbildung 3.2: Assoziativität Suchen

4 Installation

Das Programm ist aktuell nur auf Ubuntu 18.04 getestet.

4.1 Anforderungen

Um das Programm ausführen zu können wird LLVM 5.0 und CLANG 5.0 benötigt. Des weiteren muss mit dem Bash Befehl *clang* und $llvm - link$ auch selbiges Programm ausgeführt werden. Zusätzlich wird zum erstellen des Datenstroms der Ordner /tmp im Wurzelverzeichnis benötigt.

4.2 Kompilieren

Um das Programm zu kompilieren, werden *make* und *cmake* benötigt. Durch den Befehl make im Wurzelverzeichnis werden alle benötigten Daten kompiliert und das Programm mit dem Titel cSim beinhaltet alle funktionalitäten.