Latency Hiding in Tree Lookups using Out Of Order Execution

Lukas Karnowski

21. November 2017

Zusammenfassung

Moderne Prozessoren beschleunigen die Ausführung, indem sie Out-Of-Order-Execution verwenden. Werden Programme auf dieses Konzept optimiert, so ist eine deutliche Geschwindigkeitszuname zu verzeichnen. Diese Arbeit wendet die Optimierungen auf den Lookup in einem Adaptive-Radix-Tree an und ist so in der Lage, in bestimmten Fällen, die Anzahl an Lookups pro Sekunde zu verdoppeln.

1 Einleitung

Die Entwicklung von Prozessoren lässt sich mit *Moore's Law* beschreiben. Dies besagt, dass sich die Komplexität von integrierten Schaltkreisen alle 18 Monate verdoppelt. Damit einhergehend wurde früher vermutet, dass sich ebenfalls die Taktrate neuer Prozessoren regelmäßig verdoppelt, was eine Verdoppelung der Ausführungszeit zur Folge hätte. Die Realität heute zeigt jedoch, dass sich die Taktfrequenz moderner Prozessoren nicht mehr signifikant steigt. Warum also lässt sich *Moore's Law* heutzutage noch anwenden? Dies lässt sich damit erklären, dass neben der Taktfrequenz auch einige andere Mechanismen eingeführt worden sind, die Ausführungsgeschwindigkeit zu erhöhen. Darunter fällt unter anderem *Caching* und *Out-Of-Order-Execution*. Ersteres verbessert die Zugriffszeit auf häufig verwendete Bereiche im Hauptspeicher, letzteres erlaubt es, Instruktionen außerhalb ihrer Reihenfolge im Assembler Code auszuführen.

Diese Arbeit präsentiert einen verbesserten Lookup-Algorithmus für den „Adaptive-Radix-Tree“, einer Hauptspeicherindexstruktur. Dieser Algorithmus ist speziell auf Out-Of-Order-Execution und Caching ausgelegt. Dafür wird in Abschnitt 2 zunächst Hintergrundwissen über die verwendete Datenstruktur gegeben, sowie Out-Of-Order-Execution erläutert. Anschließend wird in Abschnitt 3 die Implementierung erläutert und die Ergebnisse diskutiert. Schließlich wird in Abschnitt 4 die Technik bewertet und mögliche zukünftige Arbeit beschrieben.

2 Hintergrund

In diesem Abschnitt soll dem Leser ein Überblick über die verwendeten Konzepte gegeben werden. Daher erläutern die beiden folgenden Unterkapitel einerseits die verwendete Baumstruktur *Adaptive-Radix-Tree* (ART), und andererseits das Konzept der *Out-Of-Order-Execution* (OOOE), mit der moderne Prozessoren rechenintensive Operationen zurückstellen können.

2.1 Der Adaptive-Radix-Tree

Der ART (Adaptive-Radix-Tree) ist eine Baumstruktur die speziell für effiziente Indexierung in Hauptspeicherdatenbanksystemen genutzt werden kann. Dabei verhält sie sich genau wie ein normaler Radix-Baum (manchmal auch Trie oder Präfixbaum genannt), bei denen in jedem Knoten der Präfix der darunterliegenden Schlüssel gespeichert wird. Nutzt man dies zur Speicherung von Datentypen fester Größe, so hat der daraus entstehende Radix-Baum immer eine konstante Höhe. Zum Beispiel könnte man bei der Speicherung von 64-Bit Schlüsseln jeden Schlüssel in 8 Bytes zerlegen, sodass jede Baumebene genau einem Byte entspricht und der daraus entstehende Baum genau eine Höhe von 8 besitzt (ohne die Blattknoten).

Der ART optimiert dieses Konzept indem er die einzelnen Knoten innerhalb des Baumes dynamisch vergrößert beziehungsweise verkleinert und so seine sehr effiziente Datenspeicherung ermöglicht. Des Weiteren ist der ART in der Lage, Pfade zu komprimieren und so Lookup-Zeiten zu verringern. Diese Konzepte heißen „Path-Compression" und „Lazy-Expansion" [1], sind aber für diese Arbeit nicht relevant. Ein Beispiel für einen einfachen ART ist in Abbildung 1 gezeigt. Es werden 3-stellige Wörter gespeichert. Jede Baumebene entspricht einem Zeichen und jeder Knoten ist für alle Wörter zuständig, die den Präfix besitzen, der innerhalb des Knotens notiert ist. So besitzt zum Beispiel der Wurzelknoten den Präfix des leeren Wortes ϵ, womit alle gespeicherten Wörter unterhalb dieses Knotens liegen.

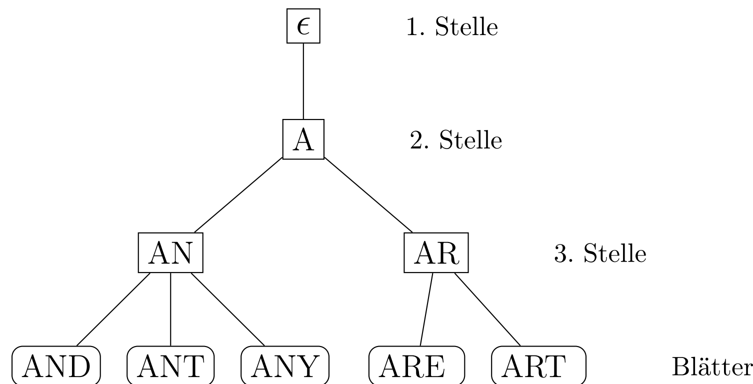

Abbildung 1: Beispielhafter ART. Speicherung von 3-stelligen Wörtern (nach [1]).

2.1.1 Knotentypen

Wie bereits genannt passt der ART die Knotengröße an die aktuelle Datenauslastung an. Dies geschieht mit Hilfe verschiedener Knotentypen, die jeweils eine feste Anzahl an Knoten beinhalten können. Insgesamt besitzt der ART vier solcher Knotentypen. Dabei wird davon ausgegangen dass ein Teilschlüssel genau ein Byte groß ist.

- **Node4**: Zuerst jeweils ein Byte für die 4 Teilschlüssel und anschließend 4 Pointer auf die nächsten Knoten.
- **Node16**: (Gleicher Aufbau wie Node4) Zuerst jeweils ein Byte für die 16 Teilschlüssel und anschließend 16 Pointer auf die nächsten Knoten.
- **Node48**: Zuerst ein Byte Array der Größe 256 und anschließend 48 Pointer auf die nächsten Knoten. Der Inhalt des Byte Arrays ist der Offset innerhalb der Pointer.

- **Node256**: Array von 256 Pointer auf die nächsten Knoten.

Für diese Arbeit ist es nicht relevant wie jeweils der Kindknoten aus den einzelnen Knotentypen extrahiert werden kann. Die Funktion dafür hat den Namen **findChild()** und wird genauer in [1] erläutert.

2.1.2 Lookup

Der Lookup im ART lässt sich nun wie folgt durchführen: In jedem Schritt wird im gerade traversierten Node ein Lookup nach dem aktuellen Teilschlüssel durchgeführt. Dies übernimmt die Funktion **findChild()** und gibt dabei den nächsten Knoten zurück. Damit startet der Algorithmus wieder erneut. Eine vereinfachte, rekursive Version in Pseudo-Code zeigt Abbildung 2.

```
1  lookup(node, key, depth):
2    if node == NULL
3      return NULL
4    if isLeaf(node)
5      if leafMatches(node, key, depth)
6        return node
7      return NULL
8    // ...
9    next = findChild(node, key[depth])
10   return lookup(next, key, depth+1)
```

Abbildung 2: Vereinfachter Lookup Algorithmus im ART (nach [1])

Anhand dieses Lookup-Algorithmus wird in Abschnitt 3 eine effizientere Version präsentiert, der die Out-Of-Order Fähigkeiten des Prozessors ausnutzt. Um ein besseres Verständnis für Out-Of-Order-Execution zu bekommen, wird im folgenden Unterabschnitt genauer darauf eingegangen.

2.2 Out-Of-Order-Execution

Moderne Prozessoren sind in der Lage, Instruktionen in anderer Reihenfolge durchzuführen, als sie im Assembler Code angegeben wurden. Dabei stellt der Prozessor sicher, dass es nach außen so wirkt, als wäre alles in der korrekten Reihenfolge durchgeführt worden. Intern schafft es der Prozessor so, die Ausführungszeit von Code zu reduzieren. Ein einfaches Beispiel ist der Ausdruck **(a+b)+(c+d)**. Der Prozessor kann bereits beginnen **(c+d)** zu berechnen, bevor **(a+b)** fertig berechnet wurde, da keine Abhängigkeit zwischen dem ersten und dem zweiten Ausdruck besteht [3]. Das dabei verwendete Konzept nennt sich Out-Of-Order-Execution (OOOE).

Bei solchen „einfachen“ arithmetischen Operationen fällt der Geschwindigkeitsunterschied kaum auf. Betrachtet man aber einen Hauptspeicherzugriff, so kann dieser deutlich länger dauern, wenn das adressierte Datum nicht in einem der Caches liegt. Schafft man es also, zwei Hauptspeicherzugriffe mit Cache-Miss hintereinander auszuführen, so müsste der Prozessor in der Lage sein, diese parallel durchzuführen. Um diese Theorie zu verifizieren, wird im folgenden Unterabschnitt ein Experiment mit verketteten Listen präsentiert.

2.2.1 Linked-List-Experiment

In diesem Experiment wird eine einfache verkettete Liste (Linked-List) traversiert. Ein Node ist dabei einfach ein Datentyp, der einen Pointer auf das nächste Element beinhaltet und einige Daten. Die Anzahl der Daten ist gerade so gewählt,

dass ein Node genau eine Cache Line von 64 Byte belegt. In Abbildung 3 ist der entsprechende Code gezeigt. Die Liste wird auf dem Heap alloziert und die Reihenfolge der einzelnen Nodes zufällig festgelegt. Die Anzahl an Elementen ist von der Größenordnung 2^{20}, sodass möglichst nur ein kleiner Teil in den Cache passt.

```
1 struct Node {
2   Node *next;
3   std::uint8_t data[56];
4 };
5 // ...
6 for (Node *curr = list; curr != nullptr; curr = curr->next) {
7   // Empty body
8 }
```

Abbildung 3: Iteration über eine einfach verkettete Liste.

Der Prozessor sollte bei diesem Code nicht in der Lage sein mittels OOOE Speicherzugriffe parallel durchzuführen, da in jeder Schleifeniteration das Ergebnis der vorherigen Iteration benötigt wird. Verifizieren lässt sich dies, indem man den Ausschnitt des Assembler-Codes in Abbildung 4 betrachtet.

```
1 0x3590: mov  (%rax),%rax
2 0x3593: test %rax,%rax
3 0x3596: jne  0x3590
```

Abbildung 4: Assembler Code zur Iteration über eine einfach verkettete Liste.

Man kann gut erkennen, dass eine Abhängigkeit zwischen der ersten Instruktion (`0x3590`) und der zweiten (`0x3593`) besteht, da zunächst das Ergebnis des Speicherzugriff in das Register `%rax` geschrieben wird und anschließend mit diesem Ergebnis die `test`-Instruktion durchgeführt wird. Diese Abhängigkeit zwischen den ersten beiden Instruktionen nennt sich *RAW*, was für *Read After Write* steht [4]. Die erste Instruktion muss also beendet werden, bevor die zweite ausgeführt werden kann.

Um nun dem Prozessor die Möglichkeit zu geben, OOOE einzusetzen, sollen nun zwei Listen parallel iteriert werden. Der Code dafür ist in Abbildung 5 gezeigt. Wichtig hierbei ist, dass beide Listen nur noch jeweils halb so viele Elemente besitzen (im Vergleich zum ersten Beispiel aus Abbildung 3). In der Summe werden aber immer noch gleich viele Elemente besucht, weshalb man die Ausführungszeit beider Experimente miteinander vergleichen kann. Der Assembler Code dazu befindet sich in Abbildung 6.

```
1 for (Node *curr1 = list1, *curr2 = list2;
2   curr1 != nullptr && curr2 != nullptr;
3   curr1 = curr1->next, curr2 = curr2->next) {
4   // Empty body
5 }
```

Abbildung 5: Iteration über zwei einfach verkettete Listen.

```
1 0x3600: mov   (%rax),%rax
2 0x3603: mov   (%rdx),%rdx
3 0x3606: test  %rax,%rax
4 0x3609: je    0x3610
5 0x360b: test  %rdx,%rdx
6 0x360e: jne   0x3600
7 0x3610: ...
```

Abbildung 6: Assembler Code zur Iteration über zwei einfach verketteten Listen.

Anhand des Assembler Codes lässt sich erkennen, dass nun keine Abhängigkeit mehr zwischen den ersten beiden Instruktionen besteht, da sie jeweils auf unterschiedliche Register zugreifen. Daher ist der Prozessor in der Lage, beide Instruktionen parallel auszuführen. Um zu erkennen, welchen Geschwindigkeitsunterschied dies ausmacht, wurde der obige Code mit Zeitmessungen versehen, deren Ergebnisse in Abbildung 8 zu sehen sind. Die X-Achse gibt dabei die Anzahl an parallel iterierten Listen an, die Y-Achse die Anzahl an besuchten Listeneinträgen pro Mikrosekunde. Die technischen Daten des Rechners, auf dem die Experimente durchgeführt worden sind, werden in Abbildung 7 gezeigt.

Prozessor	Intel® Core™ i5-5200U
Mikroarchitektur	Broadwell
L1-Cache	32KiB
L2-Cache	256KiB
L3-Cache	3 MiB

Abbildung 7: Technische Daten des Rechners, auf dem die Experimente ausgeführt wurden.

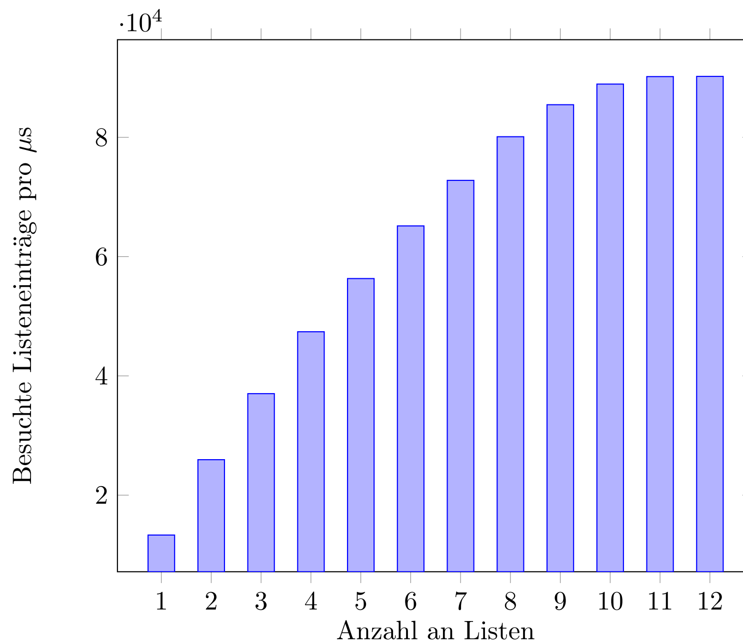

Abbildung 8: Ergebnisse des Linked-List Experiments.

Es lässt sich gut erkennen, dass die Anzahl an besuchten Listeneinträgen mit steigender Anzahl an Listen steigt, die Zuname jedoch mit der Zeit immer geringer wird, bis schließlich kaum noch eine Zuname zu verzeichnen ist.

Ein ähnliches Bild zeichnet sich ab, wenn man dasselbe Experiment auf anderen Prozessoren laufen lässt. So zeigt Abbildung 9 die Messungen von einem Intel Skylake und Sandy Bridge Prozessor.

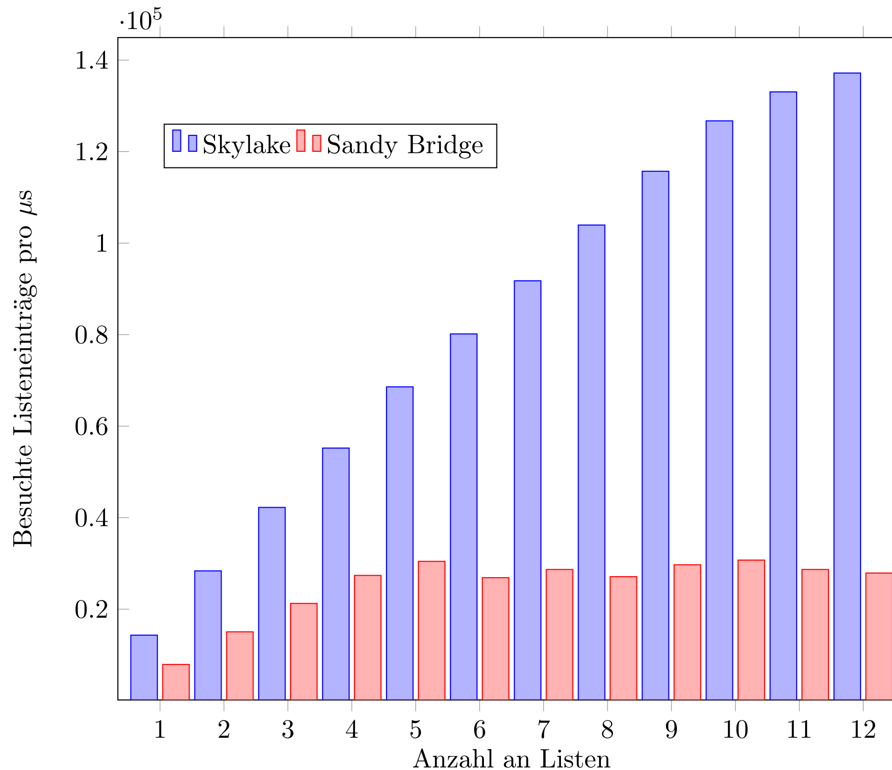

Abbildung 9: Weitere Messungen auf Intel Skylake und Sandy Bridge Prozessoren.

Interessant hierbei zu sehen ist, dass der neuere Skylake Prozessor in der Lage ist, deutlich mehr zu parallelisieren und so die Geschwindigkeit fast verzehnfacht. Der ältere Sandy Bridge Prozessor hingegen ist nur in der Lage bis zu fünf Listen gleichzeitig zu iterieren, bis keine Geschwindigkeitszuname mehr zu verzeichnen ist. Dies zeigt sehr schön die bereits in der Einleitung angesprochene Entwicklung modernerer Prozessoren, nicht mehr die Taktfrequenz zu erhöhen, sondern anderweitig die Ausführung zu beschleunigen - in diesem Fall durch Out-Of-Order-Execution.

2.2.2 Evaluation

Im vorangegangenen Unterabschnitt wurde gezeigt, dass moderne Prozessoren sehr gut Speicherzugriffe parallelisieren können. Auf den für die Experimente verwendeten Rechnern konnte die Geschwindigkeit fast verzehnfacht werden. Beachten muss man allerdings, dass die Experimente unter optimalen Bedingungen ausgeführt worden sind. Der verwendete Code wurde ausschließlich zum Test der OOOE verwendet, weshalb der Overhead durch mögliche andere Operationen nicht beachtet wurde. Nutzt man dasselbe Konzept im Kontext von Indexstrukturen, so ist zu erwarten, dass die Geschwindigkeit nicht derartig erhöht werden kann. Um dies

zu verifizieren, wird im folgenden Abschnitt die Implementierung des genannten Konzepts der OOOE anhand des Lookups im ART aufgezeigt.

3 Implementierung im Adaptive-Radix-Tree

In diesem Abschnitt wird die Optimierung auf OOOE anhand des ART beschrieben. Dabei wird zunächst die Implementierung im ART erläutert, und anschließend Messergebnisse mit den durchgeführten Testfällen präsentiert. Abschließend werden diese Ergebnisse bezüglich ihrer Praxistauglichkeit diskutiert.

3.1 Die Implementierung

In diesem Abschnitt wird die Implementierung im ART präsentiert. Die Technik nennt sich „Group-Prefetching“ [2] und führt mehrere Lookups zur gleichen Zeit aus. In Unterunterabschnitt 2.1.2 wurde bereits der Lookup Algorithmus für den ART Baum erläutert. Das Problem daran: Der Zustand eines Lookups wird dabei innerhalb der Ausführungseinheit der Funktion gespeichert, mehrere Lookups gleichzeitig sind damit nicht möglich. Um dieses Problem zu umgehen, wird für Group-Prefetching eine neue Datenstruktur eingeführt, die den Zustand eines einzelnen Lookups speichert. Die Deklaration dieser Datenstruktur befindet sich in Abbildung 10.

```
1  struct GPState {
2    std::uint8_t key[8];
3    Node *node;
4
5    unsigned depth = 0;
6    bool skippedPrefix = false;
7    bool finished = false;
8
9    GPState() : node(nullptr) {}
10   GPState(Node *node) : node(node) {}
11 };
```

Abbildung 10: Datenstruktur zur Speicherung des Zustand eines Lookups.

Die Variablen im Zustand gleichen denen aus dem ursprünglichen Lookup Algorithmus. Dabei ist `key` der Lookup-Schlüssel (ändert sich nach Initialisierung nicht mehr), `node` der Aktuelle Knoten im Lookup, `depth` wie weit der Lookup bereits fortgeschritten ist, `skippedPrefix` irrelevant für diese Arbeit und `finished` ob der Lookup abgeschlossen wurde.

Der eigentliche Lookup-Algorithmus arbeitet mit diesen Zuständen, indem er einen `std::vector` mit Zuständen erhält, und in jedem Durchlauf einer Schleife jeden der Lookups um einen Schritt voranschreiten lässt. Der Algorithmus ist in Abbildung 11 gezeigt.

```
void lookupGP(std::vector<GPState> &states) {
  std::size_t lookupCount = states.size();
  std::size_t finishedCount = 0;

  while (finishedCount < lookupCount) {
    for (auto &state : states) {
      if (state.finished) {
        continue;
      }
      // ...
      if (state.node == NULL || isLeaf(state.node)) {
        state.finished = true;
        ++finishedCount;
        continue;
      }
      // ...
      state.node = *findChild(state.node, state.key[state.depth]);
      state.depth++;
    }
  }
}
```

Abbildung 11: Gekürzter Group-Prefetching Lookup-Algorithmus.

Die Anzahl an gleichzeitigen Lookups - *Group-Size* gennannt - wird durch die Anzahl der Elemente im Zustandsvektor `states` festgelegt. Mit dieser Kenngröße kann, wie im Experiment mit verketteten Listen, der Algorithmus an die Hardware angepasst werden. Um herauszufinden, welche Auswirkung die Implementierung auf die Ausführungszeit hat, wird im folgenden Abschnitt auf die Messergebnisse eingegangen.

3.2 Messergebnisse

Um ein möglichst reales Szenario zu simulieren, wird der Test anhand eines Joins im TPC-H Schema durchgeführt [1]. Konkret wurde ein Join auf den beiden Tabellen mit den meisten Einträgen durchgeführt, also ein Join zwischen `lineitem` und `orders`. In relationaler Algebra gesprochen wird also der natürliche Join

$$\textbf{lineitem} \bowtie \textbf{orders}$$

durchgeführt. Dabei hat `lineitem` einen Fremdschlüssel auf die Tabelle `orders`, womit modelliert wird, welche Gegenstände in einer Bestellung enthalten sind. Für die Mächtigkeit der beiden beiden Tabellen gilt $|\text{lineitem}| \approx 6 \cdot 10^6$ und $|\text{orders}| = 1,5 \cdot 10^6$. Beim Join wird über die Einträge in `lineitem` iteriert und für jeden Fremdschlüssel auf `orders` ein Lookup im ART durchgeführt. Zunächst werden die Schlüssel in sortierter, aufsteigender Reihenfolge iteriert. Die Ergebnisse dieses Experiments befinden sich in Abbildung 12. Die X-Achse gibt dabei die Group-Size an, und die Y-Achse wie viele Lookups der Join pro Mikrosekunde durchführen konnte. Die rote Linie markiert die Lookups pro Mikrosekunde, die mit dem normalen Lookup erreicht werden können.

[1] `http://www.tpc.org/tpch/`

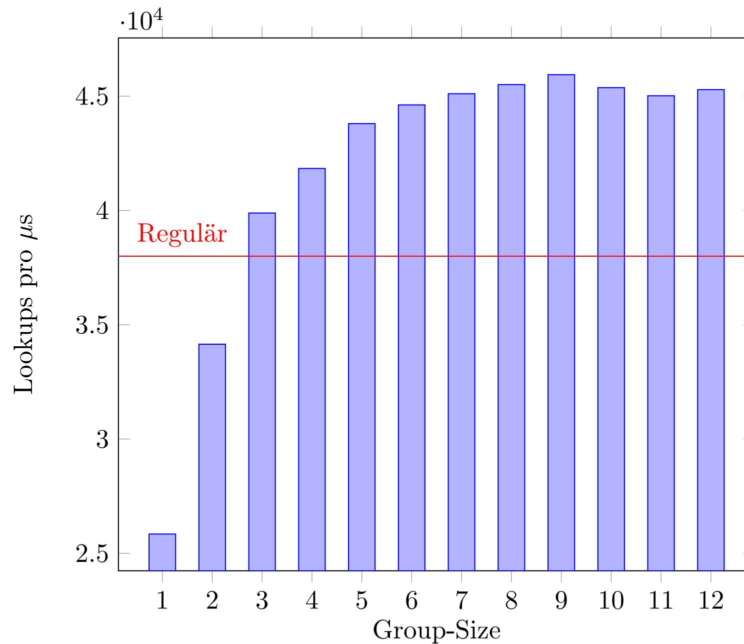

Abbildung 12: TPC-H Join in sortierter Reihenfolge.

Es lässt sich erkennen dass die Group-Prefetching Methode ab einer Group-Size von 3 bessere Ergebnisse erzielt als ein normaler Lookup, allerdings auch, dass ab einer Group-Size von 10 die Verbesserung stagniert und sogar weniger Lookups durchgeführt werden können.

In diesem Fall wurde der Join in sortierter, aufsteigender Reihenfolge durchgeführt. Dies ist in einer realen Datenbank nicht immer möglich, da die Daten nicht immer in der entsprechenden Reihenfolge vorliegen. Deshalb wurden für die Ergebnisse in Abbildung 13 die Tupel vorher randomisiert.

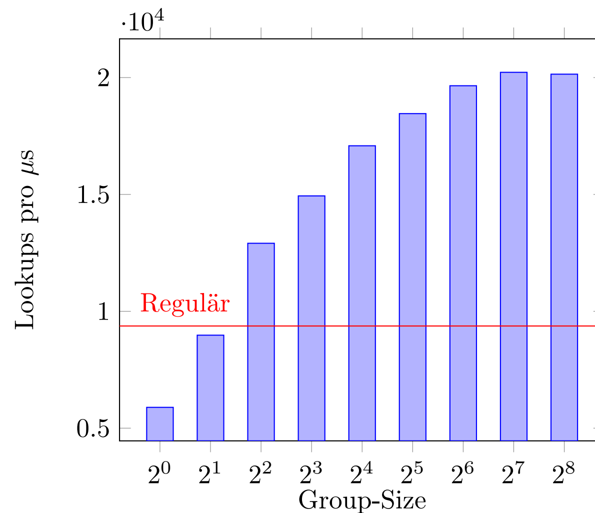

Abbildung 13: TPC-H Join in unsortierter Reihenfolge.

Interessanterweise ist in diesem Fall eine Verbesserung bis zu einer Group-Size von $2^7 = 128$ erkennen, weshalb auch eine logarithmische X-Achse für die Grafik verwendet wurde. Mit der Group-Prefetching Methode lassen sich also mehr als doppelt so viele Lookups pro Sekunde durchführen, insofern der Join in unsortierter Reihenfolge durchgeführt wird.

Im folgenden Kapitel wird nun diskutiert, ob die Methode des Group-Prefetching für die Praxis relevant ist.

3.3 Evaluation

Mit der Group-Prefetching Methode lassen sich sehr gute Ergebnisse erzielen, wenn auch nicht so gute wie im Linked-List Experiment gezeigt. Dabei konnte fast eine Verzehnfachung an Lookups pro Sekunde erreicht werden, währen im Rahmen des ART nur eine Verdoppelung der Lookups festgestellt werden konnte. Nichtsdestotrotz ist es bemerkenswert, dass eine einfache Optimierung des Lookup-Algorithmus derartige Verbesserungen in der Geschwindigkeit hervorrufen kann. Die Tatsache dass es nicht an das LinkedList-Experiment herankommt, lässt sich dadurch erklären, dass neben dem eigentlichen Speicherzugriff weitaus mehr Code benötigt wird, um einen Lookup durchzuführen. Dieser Overhead darf also nicht vernachlässigt werden.

Die Frage lautet nun, wie man automatisiert feststellen kann, in welchem Fall der normale Lookup sinnvoller ist, und in welchem Fall die Group-Prefetching Methode. Wie man gesehen hat ist im Rahmen von Joins immer dann eine Verbesserung zu verzeichnen, wenn die Group-Size mindestens einen Wert von 3 annimmt. Es ist also immer sinnvoll für einen Join die Group-Prefetching Methode zu verwenden, jedoch nicht immer klar, welchen genauen Wert die Group-Size annehmen soll. Dies hängt von der Reihenfolge ab, in der über die Join-Schlüssel iteriert werden kann. Sind diese sortiert, so wäre eine Group-Size von 9 optimal, sind die Schlüssel unsortiert, so würde etwas in der Größenordnung 2^7 die Anzahl an Lookups maximieren. Diese konkreten Werte hängen von der Hardware ab, auf der die Lookups ausgeführt werden. Es ist also nicht klar ob auf einem anderen, moderneren Prozessor die exakt gleichen Ergebnisse erzielt werden. Um also den optimalen Wert zu bestimmen, sollte das obige Experiment auf der entsprechenden Hardware durchgeführt werden.

Im Allgemeinen lässt sich festhalten, dass die Group-Prefetching Methode immer dann vorzuziehen ist, wenn schon im Voraus die Lookup-Schlüssel bekannt sind. Als weniger sinnvoll stellt sich die Methode dann heraus, wenn nur einzelne Schlüssel nachgeschlagen werden müssen, wie es beispielsweise bei einfachen SQL Abfragen der Form `SELECT * FROM tabelle WHERE attribut = Wert` der Fall ist.

4 Fazit und zukünftige Arbeit

Es ist sehr beeindruckend welchen Effekt die Optimierung auf OOOE haben kann. Deshalb ist dieses Thema durchaus relevant für zukünftige Arbeit, wie zum Beispiel die Implementierung der Technik an anderen Datenstrukturen. Des Weiteren wäre es interessant welchen Effekt „Asynchronous-Memory-Access-Chaining“ [2] auf die Implementierung im ART hat, bei dem anstatt auf alle Lookups in einer Gruppe zu warten, abgeschlossene Lookups durch neue ersetzt werden. Ferner sollte untersucht werden, warum genau für sortierte Schlüssel bei einem Join eine geringere Group-Size ausreicht, während bei unsortierten Schlüsseln größere Werte die Geschwindigkeit optimieren.

Abschließend möchte ich mich bei meinem Seminarleiter *Maximilian Schüle* bedanken, der mich auf dieses Thema aufmerksam gemacht hat, sowie bei *Timo Kersten* für die fachliche Betreuung dieser Arbeit.

Literatur

[1] Viktor Leis, Alfons Kemper, Thomas Neumann. *The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases.* 2013.

[2] Onur Kocberber, Babak Falsafi, Boris Grot. *Asynchronous Memory Access Chaining.* 2015.

[3] Agner Fog. *Optimizing software in C++.* 2017.

[4] Andrew S. Tanenbaum, Todd Austin. *Structured Computer Organization.* Pearson, 2013.