5. Übungsblatt

Ausgabe: 25. November 2019
Besprechung: 02. Dezember 2019 / 09. Dezember 2019

Einleitung

Just-In-Time (JIT) Kompilation hat sich als effiziente Verarbeitungstechnik für Datenbankanfragen etabliert. Dabei werden Anfragepläne zur Laufzeit in Maschinencode übersetzt. Zur Anfrageverarbeitung wird dann der generierte Code ausgeführt.

Mit JIT-Kompilation wird maßgeschneiderter Code je Anfrageplan erzeugt. Deshalb sind während der Ausführung keine `next()`, `nextVec()`, oder `getRelation()`-Aufrufe notwendig die den Anfrageplan interpretieren. Der Datenfluss und die Verarbeitungsschritte werden durch die Instruktionssequenz definiert. Eine Anfrage kann beispielweise wie folgt als C-Programm ausgedrückt werden:

```
--SQL
SELECT COUNT(*)
FROM   products
WHERE  p_price < 100
```

JIT →

```
1  //C
2  int query(int* p_price, int n) {
3    int count=0;
4    for(int i=0; i<n; i++) {
5      int price = p_price[i];
6      if(price < 100) {
7        count++;
8      }
9    }
10   return count;
11 }
```

Code-Generierung

Zur Code-Generierung wird der Anfrageplan systematisch durchlaufen. Während des Durchlaufs platziert jeder Operator Code-Fragmente in einem Rahmen. Sie realisieren die Funktionalität der Operatoren. Der zusammengesetzte Code realisiert den Anfrageplan.

Eine Strategie für den Plandurchlauf ist das `produce/consume` Modell [1]. Beginnend mit der Wurzel fordern Operatoren ihre Kindoperatoren durch `produce()`-Aufrufe auf Tupel zu produzieren. Erreicht die Kette den SCAN-Operator beginnt die Code-Generierung. Der SCAN-Operator produziert Tupel (vgl. Zeile 4,5,und 9 des C-Codes) und ruft die `consume()`-Funktion des Elternoperators auf. Eine Kette von `consume()`-Aufrufen verarbeitet die erzeugten Tupel bis zur Wurzel.

Aufgabe — C Query Compiler (Besprechung 02.12.)

In dieser Aufgabe soll `WeeDB` um einen C JIT-Compiler erweitert werden. Sie können dazu die Vorlage aus `WeeDB-cjit.zip` nutzen. Kompilieren Sie zunächst die Vorlage mit `make` und führen Sie `./weedb cjit` aus. Dadurch wird bereits ein Code-Rahmen generiert und ausgeführt. Betrachten Sie die Funktionalität der Klasse `CJitContext` in `CJit.h`. Experimentieren Sie mit dem Code-Generator und fügen Sie Beispielhafte Funktionalität (z.B. "Hello World") in den generierten Code ein.

Als nächstes soll ein C JIT-Compiler implementiert werden. Fügen Sie dazu in der Operator-Basisklasse `RelOperator` (`BaseOperator.h`) das Interface

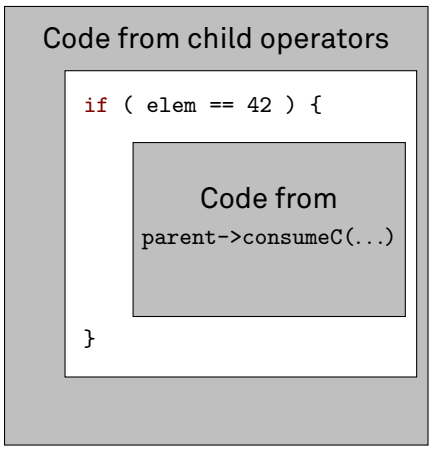
```
virtual void produceC ( CJitContext& ctx ) = 0;
virtual void consumeC ( CJitContext& ctx ) = 0;
```

hinzu. Dadurch wird die Implementation des `produceC(..)/consumeC(..)`-Interfaces in den einzelnen Operatoren erforderlich. Fügen Sie die Funktionsköpfe in `Operators.h` und die Implementationen in `OperatorsCJit.cpp` hinzu. Als Hilfestellung für die Implementation beschreiben wir hier die Code-Generierung des Selektionsoperators. Sie können sich beim Selektionsoperator und ggf. bei den anderen Operatoren daran orientieren.

Beispiel: Selektionsoperator

```
void SelectionOp::produceC (..) {
    child->produceC ( ctx );
}
void SelectionOp::consumeC (..) {
    std::ostringstream os;
    os << "if(elem";
    switch ( this->type ) {
    case PredicateType::EQUALS:
        os << "==";
        break;
    case ...
    }
    os << compareConstant << "){";
    ctx.add ( os.str() );
    if ( parent != nullptr ) {
        parent->consumeC ( ctx );
    }
    ctx.addLine ( "}" );
}
```

Code Generation →



Der Selektionsoperator arbeitet wie folgt. Die `produceC(..)`-Funktion leitet den Aufruf an den Kindoperator weiter. Das veranlasst einen Durchlauf des Kind-Teilplans der den Selektionsoperator wiederum über den Aufruf von `consumeC(..)` erreicht.

Dort wird der Code für die Selektion des Tupels in der Variable `elem` erzeugt. Zunächst wird der Kopf einer `if`-Clause mit der entsprechenden Bedingung generiert.. Dann platzieren die Elternoperatoren ihren Code durch `parent->consumeC(..)` innerhalb der `if`-Clause. Zuletzt wird die `if`-Clause geschlossen (vgl. `query.cpp` rechts).

Einführung — LLVM

Für den Einsatz von JIT-Kompiliation ist die Laufzeit der Kompilation zu Maschinencode kritisch. Mit der Ausführung von Anfragen kann in der Regel erst nach der Kompilation begonnen werden, dadurch verlängert sich die Antwortzeit um die Kompilationszeit. Eine bewährte Technik zur Verringerung der Kompilationszeit ist die Generierung von LLVM IR Code. LLVM IR Code ist eine Assembler ähnliche Sprache, die durch Ihre Ähnlichkeit zu Maschinencode verhältnismäßig schnell übersetzt werden kann.

Instruktionen

In LLVM IR Programmen werden Instruktionen als einzelne Verarbeitungsschritte verwendet. Die Instruktionen sind jeweils mit Datentypen versehen, z.B. mit `i1`, `i32` für 1-bit und 32-bit Ganzzahlen. Die Instruktionen arbeiten auf Konstanten oder Registern. Letztere haben in LLVM Bezeichner die mit dem Zeichen % anfangen.

Instruktion	Beispiel
`load`	`%i = load i32, i32* %iP` Lade den Wert von der Adresse `%iP` nach `%i`
`store`	`store i32 0,i32* %iP` Speichere den Wert `0` an der Adresse `%iP`
`alloca`	`%iP = alloca i32` Alloziere einen Wert auf dem Stack und speichere die Adresse in `%iP`
`getelementptr`	`%p = getelementptr i32, i32* %array, i32 %i` Berechne die Adresse des `%i`-ten Elements von `%array` und speichere sie in `%p`
`add`	`%sum = add i32, %a, 1` Berechne `%a`+1 und speichere das Ergebnis in `%sum`
`mul`	`%prod = mul i32, %a, 3` Berechne `%a`*3 und speichere das Ergebnis in `%prod`
`icmp`	`%isSmaller = icmp slt i32 %i, 100` Prüfe `%i`<100 und speichere das Ergebnis in `%isSmaller` (slt=signed less than)
`br`	`br i1 %isSmaller, label %next, label %end` Hat `%isSmaller` den Wert 1 springe zu `%next`, bei 0 springe zu `%end`
`ret`	`ret i32 %result` return `%result`

Basic Blocks

Basic Blocks sind Blöcke von Instruktionen und Instruktionen müssen immer in einem Basic Block stehen. Sie können als Zieladresse von 'br' Sprunginstruktionen verwendet werden. Nach einer Sprunginstruktion kann ein Basic Block nicht fortgesetzt werden. Basic Blocks stehen linksbündig mit Doppelpunkt:

```
entry:    ;start of basic block
  [...]     ;instructions
```

Single Static Assignment Form

LLVM IR ist in Single Static Assignment (SSA)-Form. Die Idee der SSA-Form basiert auf einer Maschine mit beliebig vielen Registern. Jeder neu berechnete Wert wird in ein neues Register geschrieben. Bei Zuweisungen werden die neuen Register in LLVM IR durch den Bezeichner links von dem '=' Gleichzeichen benannt. Ein Programm darf keine zwei Zuweisungen mit dem gleichen Bezeichner auf der linken Seite enthalten. Mehrfaches lesen ist unproblematisch.

Aufgabe — LLVM Beispiele (Besprechung 09.12.)

In dieser Aufgabe führen Sie Beispielprogramme mit LLVM JIT-Kompilation aus. Zunächst müssen Sie dazu LLVM installieren. `WeeDB` arbeitet mit LLVM 6.0, diese Version können Sie z.B. unter Ubuntu 18.04 über `apt` mit den Paketen `llvm-6.0` und `llvm-6.0-dev` installieren.

Beispielprogramm: `Add`

Laden Sie das Beispielprogramm `jit-add.zip` von der Veranstaltungswebseite. Kompilieren Sie das Programm mit `make` und führen Sie `./jitadd` aus.

Es wird eine simple Funktion erzeugt, die zwei ganzzahlige Parameter addiert und zurück gibt. Der unten stehenden LLVM IR code wird generiert und mit Beispielwerten ausgeführt. Schauen Sie sich den Quellcode von `createJitFunction(..)` in `jit_add.cpp` an und Versuchen Sie die Funktionsweise zu variieren. Sie können zum Beispiel `CreateAdd(..)` durch `CreateMul(..)` ersetzen oder fügen sie weitere Parameter hinzu. Das Interface zur Erzeugung der Instruktionen ist hier beschrieben:

http://llvm.org/doxygen/classllvm_1_1IRBuilder.html

```
;LLVM IR: Add two values and return result
define i32 @jit_func(i32, i32) {
entry:                    ;entry basic block
  %sum = add i32 %0, %1   ;add parameters 0 and 1, store in %sum
  ret i32 %sum            ;return %sum
}
```

Beispielprogramm: `Loop`

Laden Sie das Beispielprogramm `jit-loop.zip` von der Veranstaltungswebseite. Kompilieren Sie das Programm mit `make` und führen Sie `./jitloop` aus.

Das Programm führt eine Schleife aus, die die ersten `n` positiven Ganzzahlen addiert. Durch die SSA-Form sind Ausdrücke wie `i=i+1` nicht möglich. Um trotzdem einen Schleifenzähler zu realisieren wird dieser mit `alloca` im Speicher hinterlegt. Der Wert im Speicher wird aktualisiert, aber die Zieladresse (hier: `%pI`) bleibt unverändert. Schauen Sie sich den Quellcode von `createJitFunction(..)` in `jit_loop.cpp` an und Versuchen Sie die Funktionsweise zu variieren. Sie können zum Beispiel die Schleife so verändern, dass nur gerade Zahlen addiert werden.

```
;LLVM IR: Computes sum of first n positive integers
define i32 @jit_func(i32) {
entry:
  %pI = alloca i32          ; allocate counter,sum on stack
  %pSum = alloca i32
  store i32 0, i32* %pI     ; initialize with 0
  store i32 0, i32* %pSum
  br label %loop            ; jump to loop

loop:
  %i = load i32, i32* %pI          ; load value of counter,sum
  %sum = load i32, i32* %pSum
  %1 = add i32 %i, 1               ; increment counter, add to sum
  %2 = add i32 %sum, %i
  store i32 %1, i32* %pI           ; update stack memory
  store i32 %2, i32* %pSum
  %isLess = icmp sle i32 %1, %0            ; loop condition
  br i1 %isLess, label %loop, label %end   ; redo loop if condition

end:
  %res = load i32, i32* %pSum   ; load result value and return
  ret i32 %res
}
```

Aufgabe — LLVM Query Compiler (Besprechung 09.12.)

In dieser Aufgabe soll `WeeDB` um einen LLVM JIT-Compiler erweitert werden. Sie können dazu die Vorlage `WeeDB-jit.zip` von der Veranstaltungswebseite nutzen. Kompilieren Sie die Vorlage mit `make` und führen Sie `./weedb jit` aus. Dadurch wird bereits LLVM Code generiert und ausgeführt. Betrachten Sie die Funktionalität der Klasse `JitContext` in `Jit.h`.

Nun soll die LLVM Code-Generator Funktionalität implementiert werden. Fügen Sie dazu in der Operator-Basisklasse `RelOperator` (`BaseOperator.h`) das Interface

```
virtual void produce ( JitContext& ctx ) = 0;
virtual void consume ( JitContext& ctx ) = 0;
```

hinzu. Dadurch wird die Implementation des `produce(..)/consume(..)`-Interfaces in den einzelnen Operatoren erforderlich. Fügen Sie die Funktionsköpfe in `Operators.h` und die Implementationen in `OperatorsJit.cpp` hinzu. Die Implementation soll dabei ähnlich wie der C Code-Generator nach dem produce/consume-Modell vorgehen. Als Beispiel zeigen wir hier den LLVM IR Code der Anfrage von Seite 1 dieser Übungsaufgabe. Sie können der Klasse `JitContext` aus `Jit.h` weitere Attribute hinzufügen um es Operatoren zu ermöglichen LLVM Objekte, wie `Value*` oder `BasicBlock*` weiter zu reichen.

```
;LLVM IR for SQL query
define i32 @jitQuery(i32*,i32) {
entry:
 %iP = alloca i32         ; allocate i,count on stack
 %cntP = alloca i32
 store i32 0,i32* %iP     ; initialize with 0
 store i32 0,i32* %cntP
 br label %loop           ; jump to scan loop
loop:
 %i = load i32, i32* %iP                      ; load counter value
 %prcP = getelementptr i32, i32* %0, i32 %i ; compute element address
 %prc = load i32, i32* %prcP                  ; load element
 %less = icmp slt %prc, i32 100               ; selection condition
 br i1 %less, label doCount, label tail       ; jump to count if condition
doCount:
 %cnt = load i32, i32* %cntP    ; load current count
 %cntInc = add i32, %cnt, 1     ; add 1 to count
 store i32 cntInc,i32* %cntP    ; update count in memory
 br label %tail
tail:
 %iInc = add i32 %i, 1                  ; add 1 to i
 store i32 %iInc, i32* %iP              ; and store
 %more = icmp slt i32 %iInc, %1         ; loop condition
 br i1 %more, label %loop, label %end   ; redo loop if condition
end:
 %result = load i32, i32* cntP   ; load result
 ret i32 %result                 ; return it
}
```

Literatur

[1] T. Neumann, “Efficiently compiling efficient query plans for modern hardware,” *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.