Abgabe von Bruno Stendal, Martin Baer, Lukas Gewinner und Christian Schäfer
7. Aufgabenblatt zum Kurs

TI2: Rechnerarchitektur

von Bernadette Keßler

bis Freitag, den 06.01.2022, 10:15 Uhr.

1 Bedingte Ausführung

Beantworten Sie folgende Fragen:

1. Sprungvorhersage ist ein Konzept, um Ineffizienz im Pipelining bei Mikroprozessoren entgegenzuwirken. Befehle werden sequentiell in einen Pipeline-Kanal geladen. Erfolgt ein Sprung, werden alle davor (teils) verarbeiteten Befehle verworfen. Es existiert somit keine Effizienz. Mit der Sprungvorhersage soll ein Sprung und dessen Zieladresse so früh wie möglich erkannt werden.

2. Wikipedia: Branch predictor, und Global/Local Branch Prediction , in der VL wird das nicht erwähnt

3. Ein Vorteil von 2-Bit Automaten ggü. 1 Bit-Automaten liegt bei der Ausführung von (inneren) Schleifen. Eine Schleife nimmt immer den , abgesehen vom letzten Mal, wo sie endet. Während ein 1-Bit Automat jetzt zwar beim ersten kompletten Durchlauf der inneren Schleife immer bis auf einmal richtig lag, so würde er aber beim nächsten Anfang der inneren Schleife wieder einmal falsch liegen, da er ja zuletzt bei dieser Schleife TAKEN vorhersagte, obwohl diese Schleife dort beendet war. D.h. aufgrunddessen wird er wieder sagen NOT TAKEN, daraus folgt, dass bei inneren Schleifen immer der Anfang und das Ende falsch vorhergesagt wird. Das wird zum Problem bei weniger Durchläufen, siehe 4x4-Matrix, es ergibt sich nur eine 50 prozentige Trefferquote. 2-Bit Automaten würden hier am Ende der inneren Schleife vom STRONGLY TAKEN in den WEAKLY TAKEN Zustand wechseln, und beim nächsten Start der Schleife wieder TAKEN sagen und somit wieder zurück im Zustand STRONGLY TAKEN sein, also nur eine falsche Vorhersage pro Komplettem inneren Schleifendurchlauf. (2-Bit Automaten müssen hier quasi 2 Mal hintereinander falsch liegen um Ihre Meinung zu ändern).

4. Der Unterschied vom Hysteresis-Scheme zum Saturation-Scheme liegt darin, dass man beim S-Scheme vom Ausgangspunkt STRONG NOT TAKEN nach 2x TAKEN nur im WEAKLY TAKEN Zustand landet, statt wie beim H-Scheme im STRONGLY TAKEN. Dadurch es vorkommen das der Automat bei Schwankungen von TAKEN/NOT TAKEN immer nach einer Vorhersage seine Meinung ändert (H-Scheme MUSS immer zweimal falsch liegen um seine Meinung zu ändern). Da es aber häufiger vorkommt, dass TAKEN/NOT TAKEN über kürzere Zeiträume konstant sind, (Schleifen, im Zweifel sogar if-Abfragen), ist das H-Scheme idR. die bessere Wahl.

5. Der BTB speichert Bedingte Sprünge im Programm sowie ihre Zieladressen und die Vorhersage dafür (manchmal sogar die nächsten Befehle, je nach Prozessor). Tabellarish geordnet besteht ein Datensatz im Buffer also aus Sprungadresse, Zieladresse und Prediction-Bit/s. Wenn der Program Counter mit einer Adresse eines bedingten Sprungs

im BTB übereinstimmt, sieht man hier also schon die Zieladresse falls der Sprung genommen wird, abhängig davon, welche/s Bit/s in der Prediction steht. Diese/s Bit/s werden anhand der bisherigen Historie dieses potenziellen Sprungs gesetzt. Der BTB ist also eine weitere Form der Performancesteigerung. Völlig fehlerfrei kann dies aber nicht ablaufen, aufgrund von beispielsweise variablen Adressen.

6. Das Starten der Automaten im Zustand not takenöder ßtrongly not taken"kann dazu beitragen, die Anzahl der Fehlprognosen des BTBs zu verringern, da der Mechanismus zunächst vorhersagt, dass die Zweige nicht genommen werden. Dies kann dazu beitragen, ein Abwürgen der Pipeline zu vermeiden und die Gesamtleistung des Systems zu verbessern. Es ist im Allgemeinen nicht sinnvoll, die Automaten im Zustand takenöder ßtrongly takenßu starten, da dies dazu führen würde, dass der Mechanismus zunächst vorhersagt, dass die Zweige entnommen werden, was wahrscheinlich zu einer höheren Anzahl von Fehlvorhersagen und einer geringeren Leistung führt.

Geben Sie für folgendes Programmschema die Anzahl der Sprungvorhersagen und deren Erfolgsrate bei einem globalen 2-Bit-Prädiktor nach Hysteresis-Schema an, mit Initialisierung auf „weakly taken".

Dieses Programm besteht aus zwei Schleifen. Die äußere Schleife (mit der Bezeichnung .loop1) wird dreimal durchlaufen, und die innere Schleife (mit der Bezeichnung .loop2) durchläuft jede Iteration der äußeren Schleife zehnmal. Das bedeutet, dass die innere Schleife insgesamt 3 x 10 = 30 Mal ausgeführt wird. In diesem Programm gibt es zwei Sprunganweisungen: CMP und JL. Die CMP-Anweisung vergleicht den Wert im RBX-Register mit 10, und die JL-Anweisung (die für „jump if less" steht) bewirkt einen Sprung zum Label .loop2, wenn das Ergebnis des Vergleichs wahr ist. Das bedeutet, dass die JL-Anweisung zehnmal pro Iteration der äußeren Schleife ausgeführt wird (d. h. der Sprung wird ausgeführt). Bei einem globalen 2-Bit-Prädiktor nach dem Hystereseschema mit der Initialisierung „weakly taken" gäbe es insgesamt 30 Sprungvorhersagen (eine für jedes Mal, wenn die Anweisung JL ausgeführt wird). Die Erfolgsquote dieser Vorhersagen würde von der spezifischen Implementierung des Prädiktors und den Eigenschaften des ausgeführten Programms abhängen.

Übersetzen Sie die Befehlsfolge zunächst wie gewohnt mit einem bedingten Sprung für die IF-Verzweigung nach Assembler. Wie viele Takte sind zur Abarbeitung der Befehlsfolge nötig, wenn der Bedingungsblock ausgeführt werden soll, die Sprungvorhersage aber eine falsche Vorhersage trifft? Visualisieren Sie Ihre Lösung!

```
1  inc rdi ;
2  inc rdi ;
3  inc rdi ;
4  cmp rdi, 0 ;
5  je if_branch ;
6  jmp end      ;
7  if_branch:
8      mov rdx, 0 ;
9      div rsi, 2 ;
10     add rsi, 2 ;
11 end:
12     inc rdi ;
13     inc rdi ;
14     inc rdi ;
```

Step	IF	ID	OF	EX	WB
1.	1	NOP	NOP	NOP	NOP
2.	NOP	1	NOP	NOP	NOP
3.	NOP	NOP	1	NOP	NOP
4.	2	NOP	NOP	1	NOP
5.	NOP	2	NOP	NOP	1
6.	NOP	NOP	2	NOP	NOP
7.	3	NOP	NOP	2	NOP
8.	NOP	3	NOP	NOP	2
9.	NOP	NOP	3	NOP	NOP
10.	4	NOP	NOP	3	NOP
11.	NOP	4	NOP	NOP	3
12.	NOP	NOP	4	NOP	NOP
13.	5	NOP	NOP	4	NOP
14.	NOP	5	NOP	NOP	4
15.	NOP	NOP	NOP	NOP	NOP
16.	NOP	NOP	NOP	NOP	NOP
17.	NOP	NOP	NOP	NOP	NOP
18.	NOP	NOP	NOP	NOP	NOP
19.	NOP	NOP	NOP	NOP	NOP
20.	NOP	NOP	NOP	NOP	NOP
21.	NOP	NOP	NOP	NOP	NOP
22.	NOP	NOP	NOP	NOP	NOP
23.	NOP	NOP	NOP	NOP	NOP
24.	NOP	NOP	NOP	NOP	NOP
25.	12	NOP	NOP	NOP	NOP
26.	NOP	12	NOP	NOP	NOP
27.	NOP	NOP	12	NOP	NOP
28.	13	NOP	NOP	12	NOP
29.	NOP	13	NOP	NOP	12
30.	NOP	NOP	13	NOP	NOP
31.	14	NOP	NOP	13	NOP
32.	NOP	14	NOP	NOP	13
33.	NOP	NOP	14	NOP	NOP
34.	NOP	NOP	NOP	14	NOP
35.	NOP	NOP	NOP	NOP	14
36.			Ende.		

In Zeile 15 bis 24 ist der Pipelineflush der 10 Takte kostet.

Übersetzen Sie jetzt die Befehlsfolge unter Zuhilfenahme von Predicated Instructions. Wie viele Takte sind jetzt zur Abarbeitung der Befehlsfolge nötig? Visualisieren Sie Ihre Lösung!

Da a dreimal um 1 addiert wird vor dem Sprung und es nur in einem Fall 0 ergibt, dem Fall a = -3, kann man den bedingten Sprung je „jump equal“ in jne „jump not equal“ ändern.

```
1 inc rdi ;
2 inc rdi ;
3 inc rdi ;
```

```
4  cmp rdi, 0 ;
5  jne if_branch ;
6  je end      ;
7  if_branch:
8      inc rdi ;
9      inc rdi ;
10     inc rdi ;
11 end:
12     mov rdx, 0 ;
13     div rsi, 2 ;
14     add rsi, 2 ;
```

Step	IF	ID	OF	EX	WB
1.	1	NOP	NOP	NOP	NOP
2.	NOP	1	NOP	NOP	NOP
3.	NOP	NOP	1	NOP	NOP
4.	2	NOP	NOP	1	NOP
5.	NOP	2	NOP	NOP	1
6.	NOP	NOP	2	NOP	NOP
7.	3	NOP	NOP	2	NOP
8.	NOP	3	NOP	NOP	2
9.	NOP	NOP	3	NOP	NOP
10.	4	NOP	NOP	3	NOP
11.	NOP	4	NOP	NOP	3
12.	NOP	NOP	4	NOP	NOP
13.	5	NOP	NOP	4	NOP
14.	NOP	5	NOP	NOP	4
15.	8	NOP	NOP	NOP	NOP
16.	NOP	8	NOP	NOP	NOP
17.	NOP	NOP	8	NOP	NOP
18.	9	NOP	NOP	8	NOP
19.	NOP	9	NOP	NOP	8
20.	NOP	NOP	9	NOP	NOP
21.	10	NOP	NOP	9	NOP
22.	NOP	10	NOP	NOP	9
23.	NOP	NOP	10	NOP	NOP
24.	NOP	NOP	NOP	10	NOP
25.	NOP	NOP	NOP	NOP	10
26.			Ende.		

Geben Sie an, welche Lösung effizienter ist und unter welchen Bedingungen sich das ändern würde.
Die Ausführung unter Zuhilfenahme von Predicated Instructions ist effizienter.

2 Bubblesort

Es wurde für Debugging Zwecke mit der Rückgabe des Counters bzw. der aktuellen Vergleichsposition über rax gearbeitet, daher wird auch der modifizierte Wrapper abgegeben.

```
;Bruno Stendal, Martin Baer, Lukas Gewinner, Christian Schaefer
GLOBAL sort

SECTION .text;
;rdi = rax = max length array
;rcx = counter
;rdx = swap
;r8 = swapA
;r9 = swapB

sort:
    mov rax, rdi                    ; um den wert als debugging info zu benutzten
    .start:
        mov rcx, 0;                 ; initial den counter auf den ersten wert, nullten wert, setzten
        dec rax;                    ; den max wert um eins verringern da wir von null starten
    .check:
        cmp rax, 0;                 ; abbruch bedingung
        je .end;

        cmp rcx, rax;               ; vergleich ob das ende der zu druchsuchenden liste erreicht ist
        jl .compare;                ; sprung falls nein

        mov rcx, 0;                 ; falls ja werden das ende um eins begrenzt, siehe bubblesort, und der counter wieder zur ck gesetzt
        dec rax;
        jmp .compare;
    .compare:
        mov r8, [rsi+8*rcx];
        mov r9, [rsi+8*(rcx+1)];
        cmp r8, r9;                 ; vergleich zwischen werta und wertb
        jg .greater;                ; falls nicht getauscht wird
        jl .swapped;                ; falls getauscht wird
    .greater:
        mov [rsi+8*rcx], r9;        ; werta ersetzt wertb
        mov [rsi+8*(rcx+1)], r8;    ; wertb ersetzt werta
        jmp .swapped;
    .swapped:
        inc rcx;                    ; counter um eins erh hen um das n chste paar zu checken
        jmp .check;
    .end:
        ret;                        ; funktiosende
```

```c
#include <inttypes.h>
#include <limits.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define BOUND 100

extern uint64_t sort(uint64_t len, int64_t a[len]);

static inline void printArray(int64_t *to_show, uint64_t len, uint64_t
    len_func)
{
   for (uint64_t i = 0; i < len; i++) {
      if (i == 0) { // erstes Element
         printf("Array: %" PRId64 ", ", to_show[i]);
      } else if (i == (len - 1)) { // letztes Element
         printf("%" PRId64 "\n", to_show[i]);
         printf("Return value: %ld \n", len_func);
      } else {
         printf("%" PRId64 ", ", to_show[i]);
      }
   }
}

int main(int argc, char *argv[])
{
   if (argc < 2) {
      fprintf(stderr, "Not enough arguments!\n");
      printf("Usage: ./bubblesort <len>\n");
      return 1;
   }

        /* Fail if
         *  1. string is empty or
         *  2. there were trailing characters or
         *  3. the converted value is out of range
         */
        char *endptr;
        const uint64_t len = strtoull(argv[1], &endptr, 10);
        if (argv[1][0] == '\0' || *endptr != '\0' ||
            (len == ULONG_MAX && errno == ERANGE)) {
                fprintf(stderr, "Invalid Argument: \"%s\"\n", argv[1])
   ;
                return 2;
        }

   if (len < 2) {
      fprintf(stderr, "Length must be at least 2.\n");
      return 3;
   }

```

```
52      srand(time(NULL));
53      int64_t to_sort[len];
54      for (uint64_t i = 0; i < len; i++) {
55          to_sort[i] = rand() % BOUND;
56          // int8_t neg = rand() % 2;
57          // if(neg) {
58          //     to_sort[i] = to_sort[i] * -1;
59          // }
60      }
61
62      uint64_t lengthAfterSort = 0;
63      printArray(to_sort, len, len);
64      lengthAfterSort = sort(len, to_sort);
65      printArray(to_sort, len, lengthAfterSort);
66  }
```