Теневые регистры
Предсказатель перехода

Денисов А. А.

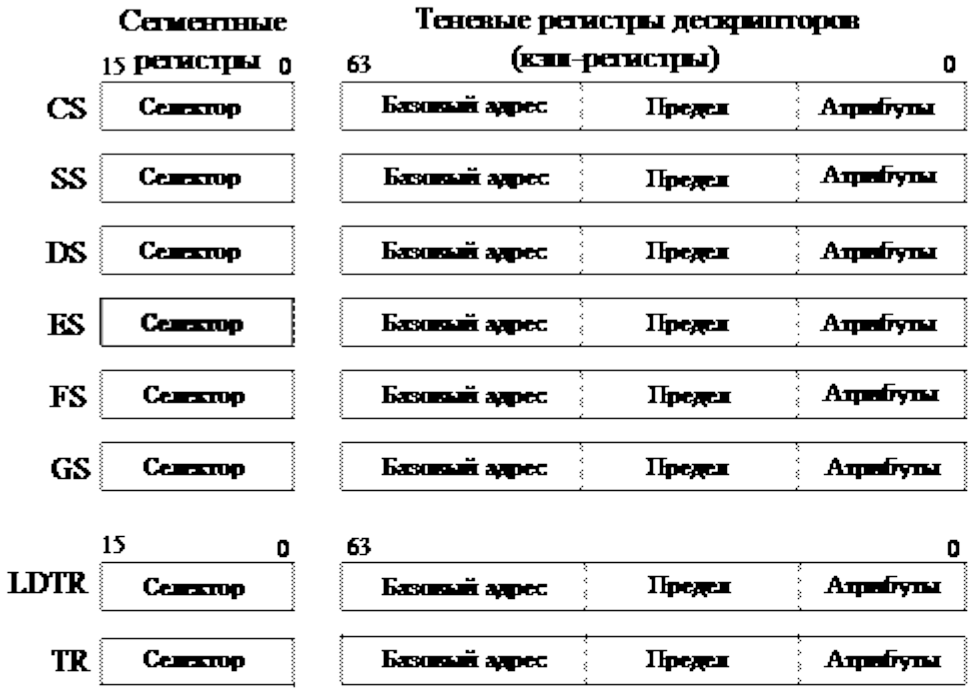
Теневые регистры

x86

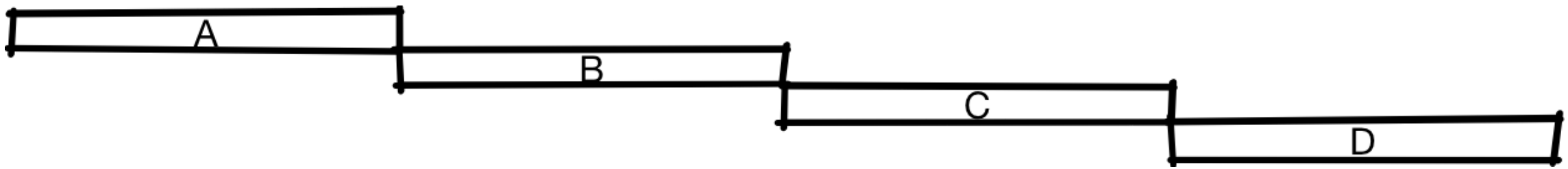
Обращение к 1 байту памяти требует:
обращение к двух 8-байтовым дескрипторам (в GDT и LDT)
и выполнении двух 32-битных сложений.

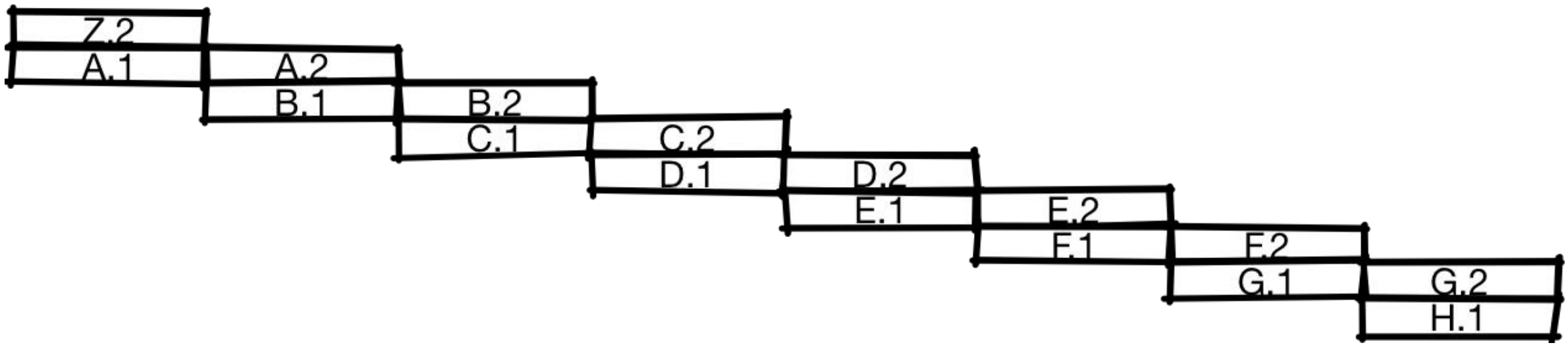
1. Получение адреса LDT в GDT (складывая базы GDT из GDTR и базы из дескриптора LDT из LDTR)
2. Получение окончательного адреса складывая начало соответствующего сегмента и эффективного адреса

Но на самом деле обращения к памяти происходят гораздо чаще, чем изменение содержимого сегментых регистров и переключение задач. Можно загрузить дескриптор в процессор вместе с селектором и не делать обращение в ОЗУ для формирования адреса.



Предсказатель перехода


A
B
C
D


Z.2
A.1
A.2
B.1
B.2
C.1
C.2
D.1
D.2
E.1
E.2
F.1
F.2
G.1
G.2
H.1

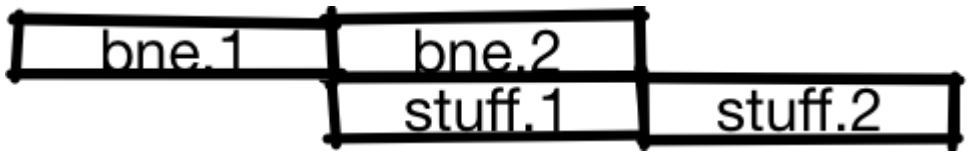
```
if (x == 0) {
  // Do stuff
} else {
  // Do other stuff (things)
}
  // Whatever happens later
```

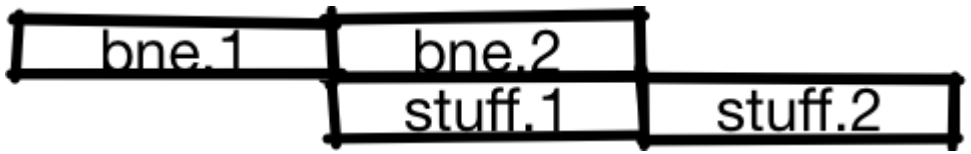
```
branch_if_not_equal x, 0, else_label
// Do stuff
goto end_label
else_label:
// Do things
end_label:
// whatever happens later
```

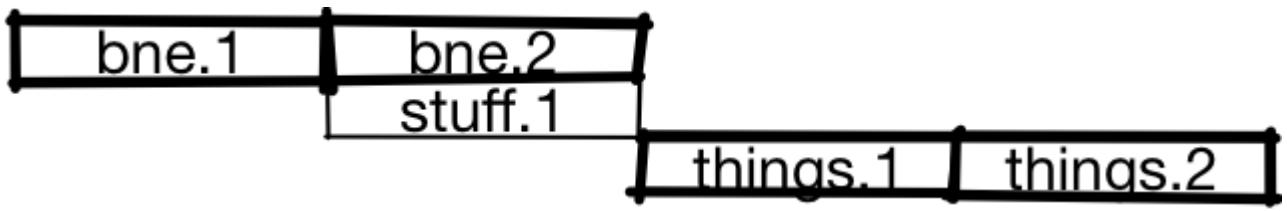
```
branch_if_not_equal x, 0, else_label
???
```

```
branch_if_not_equal x, 0, else_label
// Do stuff
```

```
branch_if_not_equal x, 0, else_label
// Do things
```


bne.1
bne.2
stuff.1
stuff.2


bne.1
bne.2
stuff.1
stuff.2


bne.1
bne.2
stuff.1
things.1
things.2

SPECint

20 ступенчатый конвейер

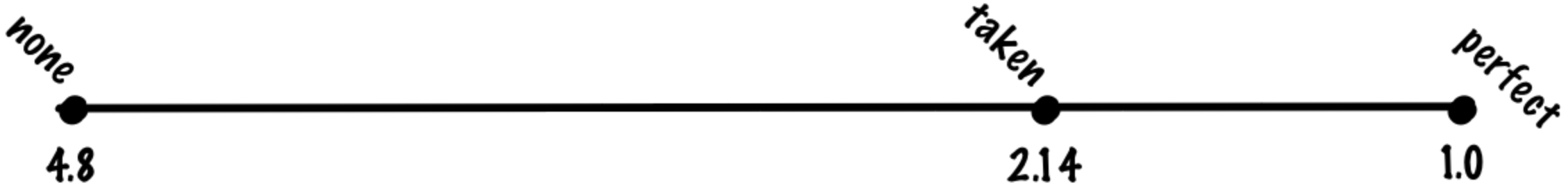
20% ветвлений

80% обычных операций

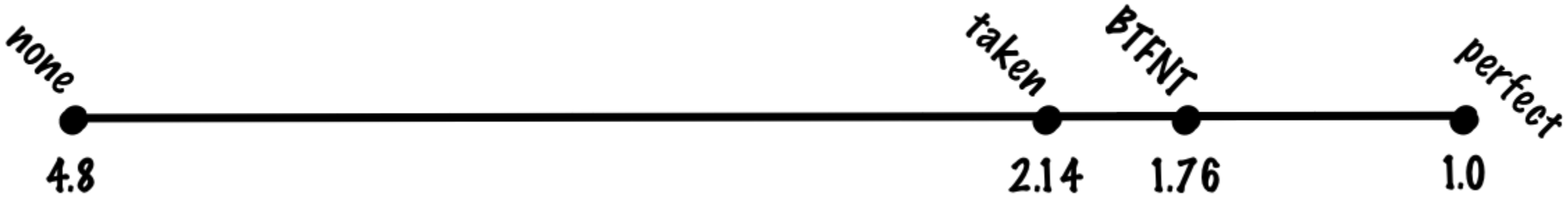
branch_pct * 1 + non_branch_pct * 20 = 0.8 * 1 + 0.2 * 20 = 0.8 + 4 = 4.8

0.8 * 1 + 0.2 * 1 = 1

Берём все переходы
(например, для циклов)

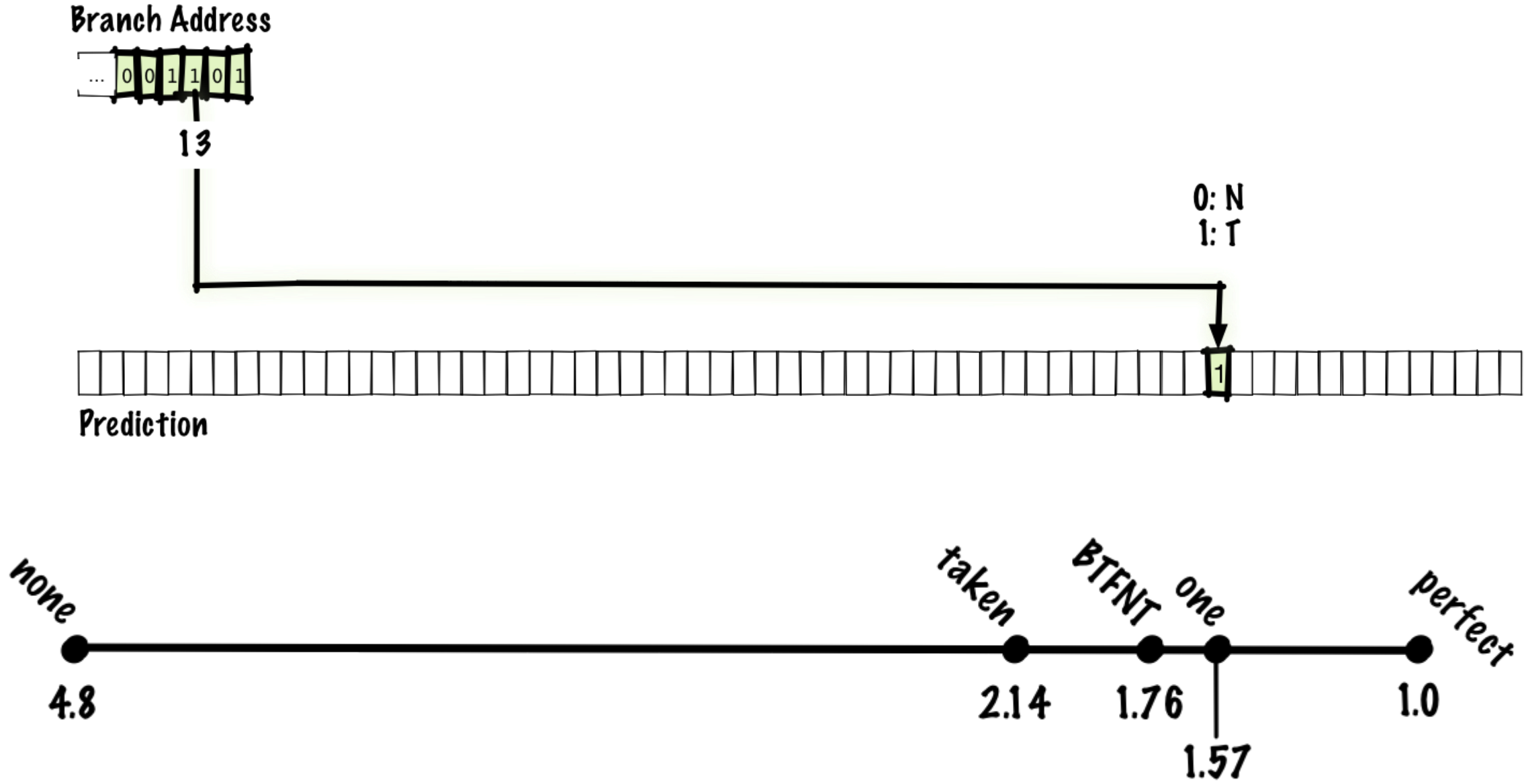


Берём переходы назад, не берём переходы вперёд (BTFNT)
(а компиляторы подстроятся под нас)



Это были статические методы предсказания переходов.

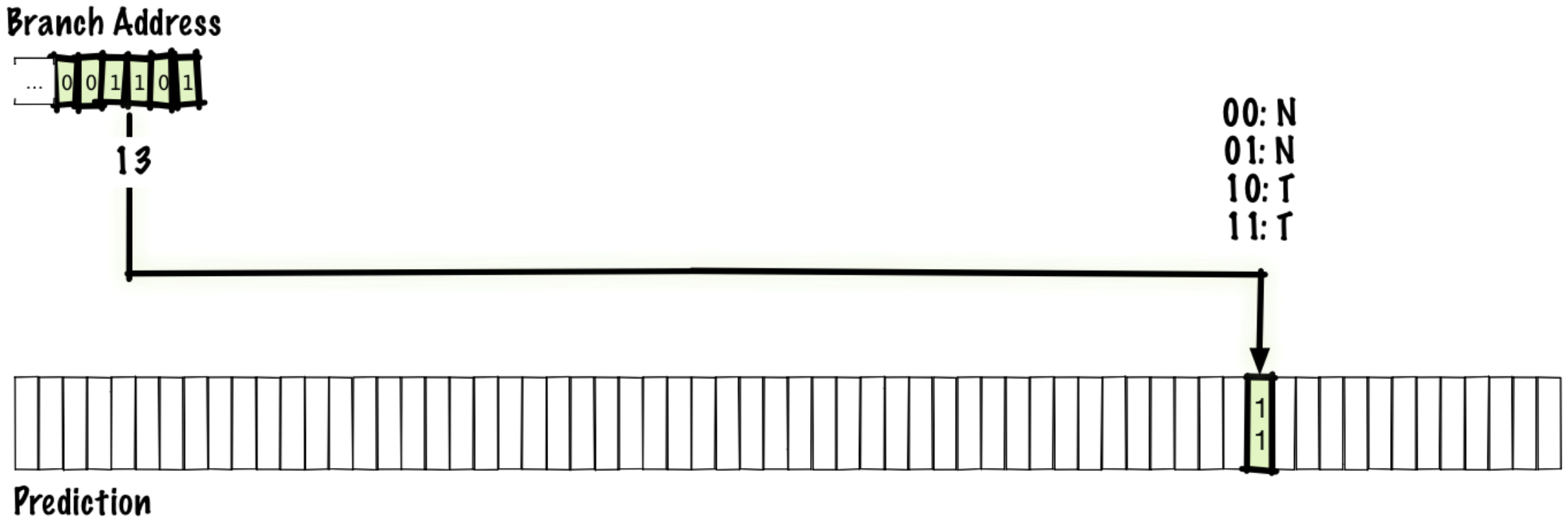
```
if (flag) {
  // things
  }
```


Branch Address
... 0 0 1 1 0 1
13
0: N
1: T
1
Prediction
none
taken
BTFNT
one
perfect
4.8
2.14
1.76
1.57
1.0

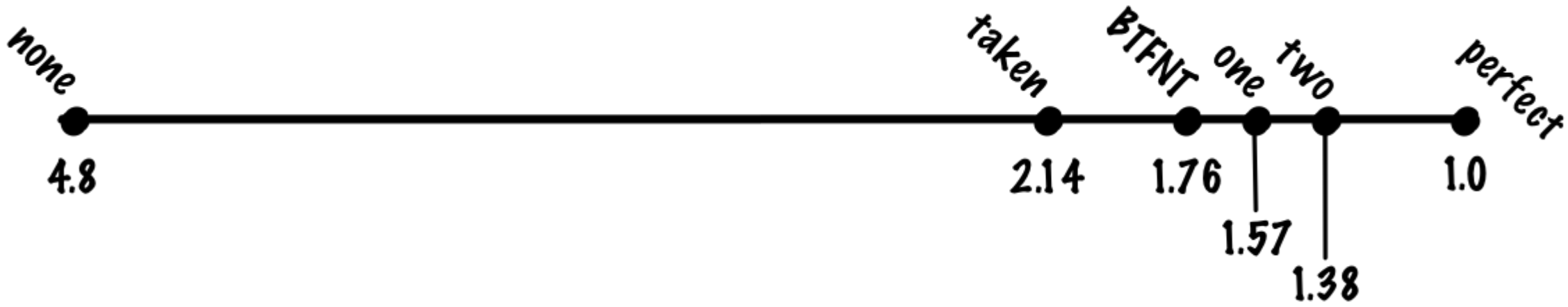
TTTTTTTT NNNNNNN

TTTNTTT

00: predict Not
01: predict Not
10: predict Taken
11: predict Taken

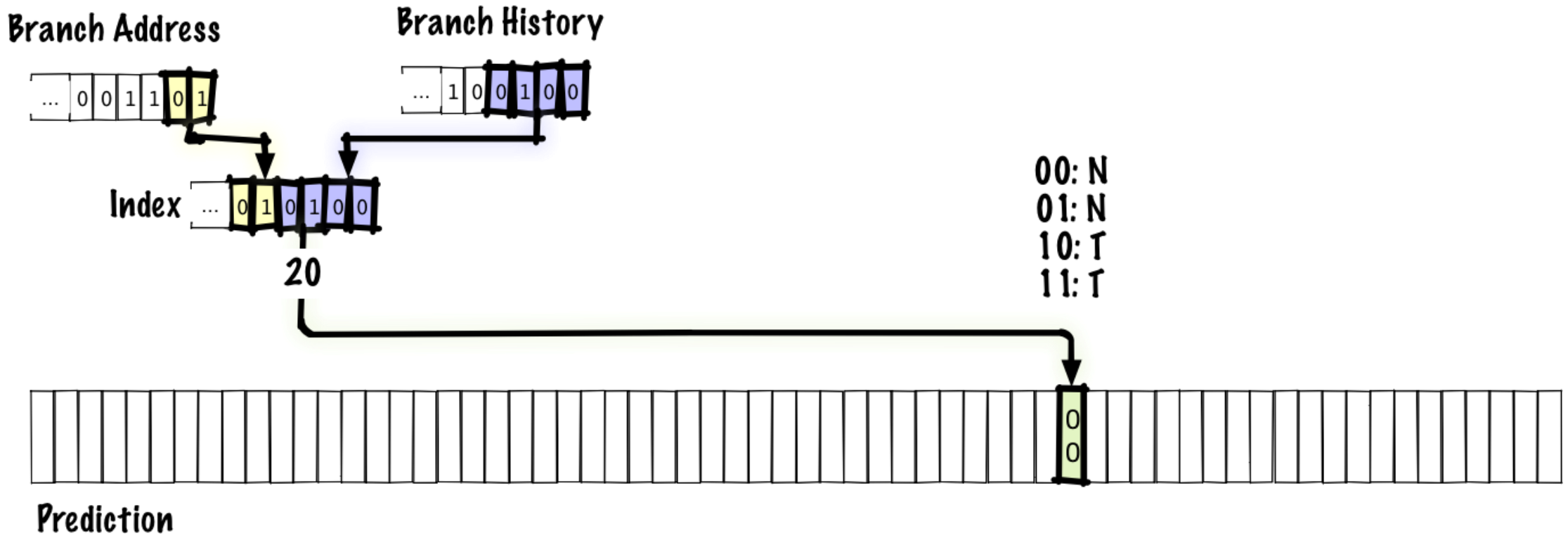


- LLNL S-1 (1977)
- CDC Cyber? (начало 80-х)
- Burroughs B4900 (1982)
- Intel Pentium (1993)
- PPC 604 (1994)
- DEC EV45 (1993)
- DEC EV5 (1995)
- PA 8000 (1996)

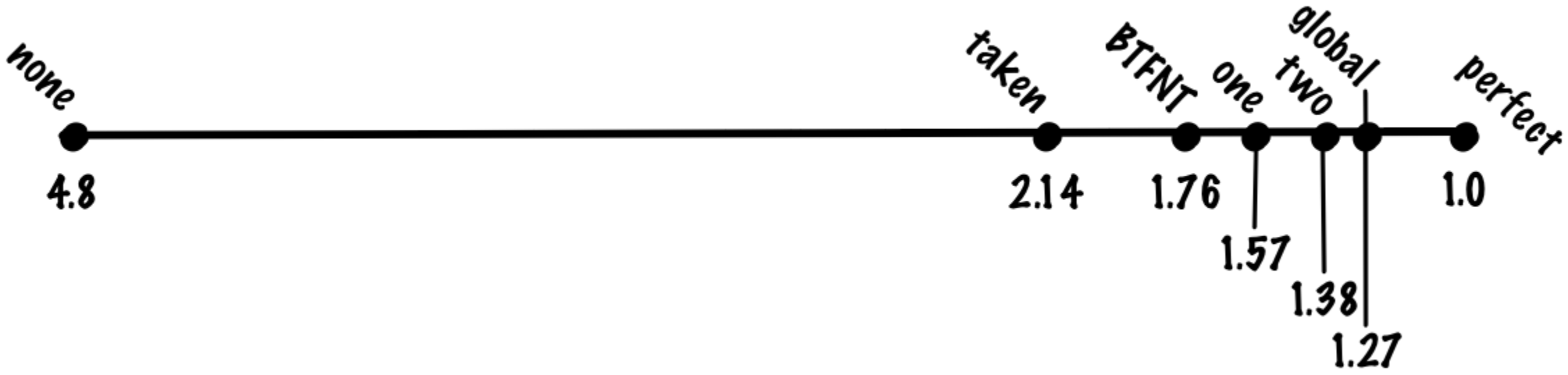


```
for (int i = 0; i < 3; ++i) {
  // code here.
  }
```

TTTNTTTNTTTN



С этой схемой мы можем добиться точности 93%,
что соответствует 1,27 цикла на инструкцию.



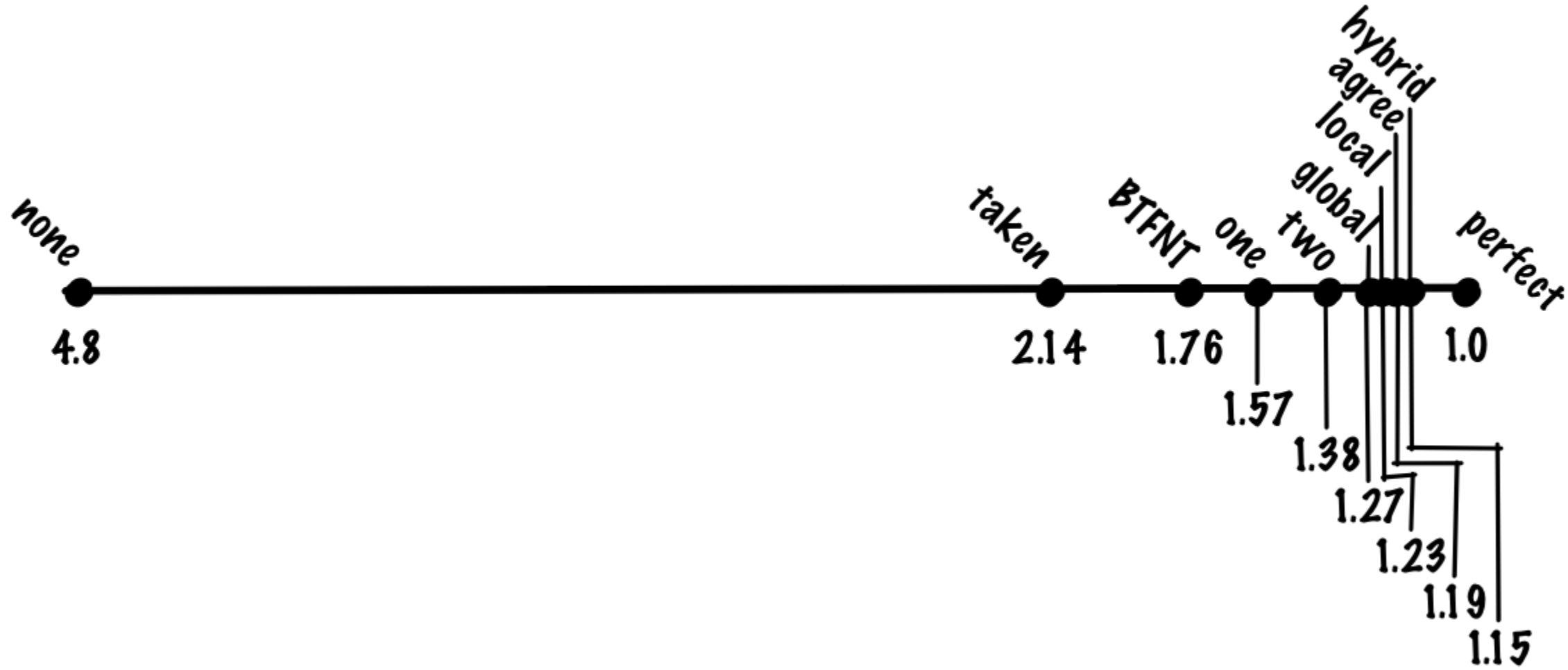
```
for (i=0; i<N; i++)
  for (j=0; j<N - (i+1); j++)
    if (a[j] > a[j+1])
      swap(a[j], a[j+1]);
```

> x3

```
for (i=0; i<N-1; i++)
    for (j=i; j>=0; j--)
        if (a[j] > a[j+1])
            swap(a[j], a[j+1]);
```

Количество переходов (ветвлений, branches): 1.0 vs. 1.0 ($\cdot 10^{10}$)

Ошибок предсказателя переходов
(mispredicted branches): **0.16 vs. 0.00009** ($\cdot 10^{10}$)


none
4.8
taken
2.14
BTFNT
1.76
one
1.57
two
1.38
global
1.27
local
1.23
agree
1.19
hybrid
1.15
perfect
1.0

Дальше результаты задачи: Алгоритм Дейкстры.

Дан взвешенный граф без отрицательных циклов.
Необходимо найти кратчайший (по весу) путь от одной вершины до всех остальных.

Алгоритм – стандартный жадный, за O(n^2).
Для усложнения задачи всегда будем генерировать полный граф со случайными весами.

Из оптимизаций :

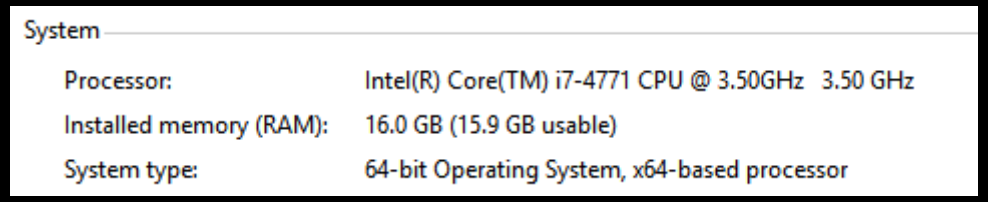
- Автоматические. Использовалась библиотека openmp. Распараллелены все внутренние циклы алгоритма. Каждому потоку выделен сплошной кусок массива, за который он отвечает, чтобы реже получать cache-miss’ы.
- Ручные. Изменил код там, где была возможность не делать лишние обращения в память, а сохранить предыдущее значение в локальную переменную. В надежде, что компилятор будет использовать для хранения этого значения регистры (не использовал в однопоточной версии, там осталось «наивное» написание кода). Стабильное, но незначительное ускорение (по сравнению с параллельной версией без этой оптимизации) наблюдалось.

Корректность кода проверяется сравнением результатов параллельной и однопоточной версией программы.

https://github.com/SilentCatVallis/parallelThings/blob/master/dijkstra.c

x64 Native Tools Command Prompt for VS 2017

```
C:\src\C++\Dijkstra>dijkstra.exe 1000
Generating time: 0.020877
8 threads calculating time: 0.009385
Single thread calculating time: 0.007954
Everything is OK
C:\src\C++\Dijkstra>
C:\src\C++\Dijkstra>dijkstra.exe 10000
Generating time: 2.359998
8 threads calculating time: 0.208919
Single thread calculating time: 0.618735
Everything is OK
C:\src\C++\Dijkstra>
C:\src\C++\Dijkstra>dijkstra.exe 20000
Generating time: 10.025747
8 threads calculating time: 0.726979
Single thread calculating time: 2.268605
Everything is OK
C:\src\C++\Dijkstra>
C:\src\C++\Dijkstra>dijkstra.exe 25000
Generating time: 15.460645
8 threads calculating time: 1.087109
Single thread calculating time: 3.480752
Everything is OK
C:\src\C++\Dijkstra>dijkstra.exe 30000
Generating time: 22.696993
8 threads calculating time: 1.624991
Single thread calculating time: 4.990202
Everything is OK
C:\src\C++\Dijkstra>
C:\src\C++\Dijkstra>dijkstra.exe 32000
Generating time: 26.477817
8 threads calculating time: 1.721579
Single thread calculating time: 5.622224
Everything is OK
C:\src\C++\Dijkstra>
C:\src\C++\Dijkstra>dijkstra.exe 35000
Generating time: 30.898754
8 threads calculating time: 2.034474
Single thread calculating time: 6.707171
Everything is OK
C:\src\C++\Dijkstra>dijkstra.exe 37000
Generating time: 34.815227
8 threads calculating time: 3.301897
Single thread calculating time: 7.991794
Everything is OK
C:\src\C++\Dijkstra>dijkstra.exe 40000
Generating time: 43.121497
8 threads calculating time: 4.674033
Single thread calculating time: 9.019711
Everything is OK
C:\src\C++\Dijkstra>99% memory, swap ;(
```



Не смотря на то, что система говорит что у неё 8 ядер, на самом деле их 4 + гипертрединг. И по моему опыту настоящий прирост x2 из гипертрединга невозможно получить даже близко (по крайне мере на задачах, не написанных специально под его возможности (маркетинговый ход)).

Как видно по результатам запуска программы с разным числом вершин в графе (выделено желтым), параллельная реализация быстрее примерно в 3 раза, чем однопоточная (хорошо прослеживается в тестах до 35000 вершин включительно). Начиная с 37000 вершин система загоняет программу в своп (потребление памяти под 100%, освобождения памяти после завершения программы ждал минуту). И ускорение становится меньше, чем в 3 раза (последовательный доступ к памяти не так сильно страдает от свопа, как случайный, который чаще случается в параллельной версии).