Good Predictions Are Worth a Few Comparisons

Carine Pivoteau

with Nicolas Auger and Cyril Nicaud

LIGM - Université Paris-Est-Marne-la-Vallée

April 2016
Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

\[
\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

\[
\begin{align*}
\text{min} &= 5 \\
\text{max} &= 5
\end{align*}
\]
Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

5 1 4 3 6 0 2 8 7 9

min = 5
max = 5
Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

\[
\begin{align*}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
1 < \text{min} \quad & \quad 1 > \text{max} \\
\text{min} = 1 \\
\text{max} = 5
\end{align*}
\]
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

5  1  4  3  6  0  2  8  7  9

min = 1
max = 5

4 < min ?
4 > max ?
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

```
5  1  4  3  6  0  2  8  7  9
```

$3 < \text{min}$ ? 

$3 > \text{max}$ ?

$\text{min} = 1$

$\text{max} = 5$
Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

\[
\begin{array}{cccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

\[
\begin{array}{c}
6 < \text{min} \ ? \\
6 > \text{max} \ ? \\
\text{min} = 1 \\
\text{max} = 5 \\
\end{array}
\]
Find both the min. and the max. of an array of size $n$.

**Naive Algorithm:**

5 1 4 3 6 0 2 8 7 9

6 < min ?

6 > max ?

min = 1

max = 6
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

```
5  1  4  3  6  0  2  8  7  9
```

```
min = 1
max = 6
0 < min ? 0 > max ?
```
Find both the min. and the max. of an array of size \( n \).

Naive Algorithm:

\begin{verbatim}
5  1  4  3  6  0  2  8  7  9
\end{verbatim}

\[ \min = 0 \]
\[ \max = 6 \]
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

5 1 4 3 6 0 2 8 7 9

\[
\begin{align*}
2 &< \text{min} \ ? \\
2 &> \text{max} \ ?
\end{align*}
\]

\[
\begin{align*}
\text{min} & = 0 \\
\text{max} & = 6
\end{align*}
\]
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

\begin{array}{cccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 \\
\end{array}

\begin{array}{cccc}
8 & 7 & 9 \\
\end{array}

\begin{align*}
\text{min} &= 0 \\
\text{max} &= 6
\end{align*}

$8 < \text{min} \ ?$  $8 > \text{max} \ ?$
Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

$$5 \ 1 \ 4 \ 3 \ 6 \ 0 \ 2 \ 8 \ 7 \ 9$$

- $8 < \min$ ?
- $8 > \max$ ?

$\min = 0$
$\max = 8$
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

\[
\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

\[
\begin{array}{c}
7 < \text{min} \ ? \\
7 > \text{max} \ ?
\end{array}
\]

min = 0
max = 8
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

5  1  4  3  6  0  2  8  7  9

$min = 0$
$max = 8$
A case study

Find both the min. and the max. of an array of size $n$.

Naive Algorithm:

\[
5 \ 1 \ 4 \ 3 \ 6 \ 0 \ 2 \ 8 \ 7 \ 9
\]

\[
\text{min} = 0 \\
\text{max} = 9
\]
Find both the min. and the max. of an array of size $n$.

Naive Algorithm: $2n$ comparisons

\[
\begin{array}{ccccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

\[
\begin{array}{ccccccccccc}
9 < \text{min} \ ? & 9 > \text{max} \ ? \\
\end{array}
\]

min = 0

max = 9

Can we do better?
Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:**

```
5  1  4  3  6  0  2  8  7  9
```

$\text{min} = 5$

$\text{max} = 5$
A case study

Find both the min. and the max. of an array of size \( n \).

**Optimized Algorithm:**

\[
\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

\[
\begin{align*}
\text{min} &= 5 \\
\text{max} &= 5 \\
5 &< 1 ? \\
1 &< \text{min} ? \\
5 &> \max ?
\end{align*}
\]
A case study

Find both the min. and the max. of an array of size $n$.

Optimized Algorithm:

\[
\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

$\min = 1$

$\max = 5$

$5 < 1 ?$

$1 < \min ?$

$5 > \max ?$
Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:**

```
5  1  4  3  6  0  2  8  7  9
min = 1
max = 5
```
Find both the min. and the max. of an array of size $n$.

Optimized Algorithm:

$$\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}$$

$$\begin{array}{l}
\text{min} = 1 \\
\text{max} = 5 \\
6 \leq 0 ? \\
0 < \text{min} ? \\
6 > \text{max} ?
\end{array}$$
Find both the min. and the max. of an array of size $n$.

Optimized Algorithm:

\[\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}\]

- $6 \leq 0$ ?
- $0 < \text{min}$ ?
- $6 > \text{max}$ ?

$\text{min} = 0$

$\text{max} = 6$
Find both the min. and the max. of an array of size $n$.

Optimized Algorithm:

\[
\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

- $2 < 8$?
- $2 < \text{min}$?
- $8 > \text{max}$?

$\text{min} = 0$

$\text{max} = 6$
Find both the min. and the max. of an array of size $n$.

### Optimized Algorithm:

\[
\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & \text{2} & \text{8} & 7 & 9 \\
\end{array}
\]

- $2 < 8$ ?
- $2 < \text{min}$ ?
- $\text{8} > \text{max}$ ?

- $\text{min} = 0$
- $\text{max} = 8$
Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:**

```
5  1  4  3  6  0  2  8  7  9
min = 0
max = 8
7 < 9 ?
7 < min ?
9 > max ?
```
Find both the min. and the max. of an array of size $n$.

Optimized Algorithm:

$\text{min} = 0$
$\text{max} = 9$
A case study

Find both the min. and the max. of an array of size $n$.

Optimized Algorithm: $3n/2$ comparisons (optimal)

\[
\begin{array}{cccccccccc}
5 & 1 & 4 & 3 & 6 & 0 & 2 & 8 & 7 & 9 \\
\end{array}
\]

\[
\begin{array}{c}
\text{min} = 0 \\
\text{max} = 9 \\
7 < 9 ? \\
7 < \text{min} ? \\
9 > \text{max} ?
\end{array}
\]
Find both the min. and the max. of an array of size $n$.

**Optimized Algorithm:** $3n/2$ comparisons (optimal)

**Naive Algorithm:** $2n$ comparisons
Find both the min. and the max. of an array of size $n$.

Optimized Algorithm: $3n/2$ comparisons (optimal)

Naive Algorithm: $2n$ comparisons

In practice, on uniform random data?
A case study

Find both the min. and the max. of an array of size $n$.

- in C,
- using `gcc -O0`,
- random integers
A case study

Find both the min. and the max. of an array of size $n$.

- in C,
- using `gcc -O0`,
- random integers
What “really” happens in the processor...

optimized min/max search

```c
// RAND_ARRAY: an array of length N
// filled with random integers

min = RAND_ARRAY[0];
max = RAND_ARRAY[0];
for(i=0; i<N; i+=2){ //assume N is even
    a1 = RAND_ARRAY[i];
    a2 = RAND_ARRAY[i+1];
    if (a1 < a2) {
        if (a1 < min) min = a1;
        if (a2 > max) max = a2;
    }
    else {
        if (a2 < min) min = a2;
        if (a1 > max) max = a1;
    }
}
```
sample of assembly code \((\texttt{gcc -O0})\)

\begin{verbatim}
mov esi, dword ptr [rbp - 60]
cmp esi, dword ptr [rbp - 64]
jge LBB2_8

mov eax, dword ptr [rbp - 60]
cmp eax, dword ptr [rbp - 12]
jge LBB2_5

mov eax, dword ptr [rbp - 60]
mov dword ptr [rbp - 12], eax

LBB2_5:
    mov eax, dword ptr [rbp - 64]
cmp eax, dword ptr [rbp - 16]
jle LBB2_7
...
\end{verbatim}
What “really” happens in the processor...

sample of assembly code (`gcc -O0`)

```assembly
mov esi, dword ptr [rbp - 60]
cmp esi, dword ptr [rbp - 64]
jge LBB2_8

mov eax, dword ptr [rbp - 60]
cmp eax, dword ptr [rbp - 12]
jge LBB2_5

mov eax, dword ptr [rbp - 60]
mov dword ptr [rbp - 12], eax
LBB2_5:
mov eax, dword ptr [rbp - 64]
cmp eax, dword ptr [rbp - 16]
jle LBB2_7
...
```

- Each instruction can be decomposed:
  ```
  IF   ID   EX   MEM   WB
  ```

- Most modern processors are pipelined

- Instructions are parallelized

simple 5 stages pipeline:
What “really” happens in the processor...

sample of assembly code (gcc -O0)

mov esi, dword ptr [rbp - 28]
cmp esi, dword ptr [rbp - 32]
jge LBB2_8

mov eax, dword ptr [rbp - 28]
cmp eax, dword ptr [rbp - 12]
jge LBB2_5

mov eax, dword ptr [rbp - 32]
mov dword ptr [rbp - 12], eax
LBB2_5:
mov eax, dword ptr [rbp - 32]
cmp eax, dword ptr [rbp - 16]
jle LBB2_7

mov eax, dword ptr [rbp - 32]
mov dword ptr [rbp - 16], eax
LBB2_7:
jmp LBB2_14

mov eax, dword ptr [rbp - 32]
cmp eax, dword ptr [rbp - 12]
jge LBB2_10

mov eax, dword ptr [rbp - 32]
mov dword ptr [rbp - 12], eax
LBB2_10:
mov eax, dword ptr [rbp - 28]
cmp eax, dword ptr [rbp - 16]
jle LBB2_14

mov eax, dword ptr [rbp - 28]
mov dword ptr [rbp - 16], eax
LBB2_14:

mov eax, dword ptr [rbp - 4]
add eax, 2
mov dword ptr [rbp - 4], eax

Each instruction can be decomposed:

IF ID EX MEM WB

Most modern processors are pipelined

Instructions are parallelized

simple 5 stages pipeline:
Branch prediction

Branch predictors are used to avoid stalls on branches!

- Conditional instructions (such as the “if” statement) yield branches in the execution of a program.
- A **misprediction** can be quite **expensive**!
- The **branch predictor** will guess which branch will be **taken** (T) or not (NT).
- Different schemes: static, **dynamic**, local, global,...
Branch predictors are used to avoid stalls on branches!

1-bit predictor:

```
not taken  
 Taken  
 taken  
 not taken
```

*Computer Architecture: A Quantitative Approach (5th ed.), Hennessy & Patterson*
Branch predictors are used to avoid stalls on branches!

2-bit predictor:
Branch prediction

Branch predictors are used to avoid stalls on branches!

2-bit predictor:
Branch prediction

Branch predictors are used to avoid stalls on branches!

Global (or mixed) predictor:

\[
\begin{array}{c|c}
\hline
& \ell \\
\hline
0000\ldots00 & \circlearrowleft \circlearrowright \\
0000\ldots01 & \circlearrowleft \circlearrowright \\
\ldots & \circlearrowleft \circlearrowright \\
1111\ldots11 & \circlearrowleft \circlearrowright \\
\hline
\end{array}
\]

Computer Architecture: A Quantitative Approach (5th ed.), Hennessy & Patterson
Branch predictors are used to avoid stalls on branches!

- Conditional instructions (such as the “if” statement) yield branches in the execution of a program.

- A misprediction can be quite expensive!

- The branch predictor will guess which branch will be taken (T) or not (NT).

- Different schemes: static, dynamic, local, global, ...

- Min and max search is very sensitive to branch prediction...

*Computer Architecture: A Quantitative Approach (5th ed.), Hennessy & Patterson*
Branch prediction

Branch predictors are used to avoid stalls on branches!

- Conditional instructions (such as the “if” statement) yield branches in the execution of a program
- A misprediction can be quite expensive!
- The branch predictor will guess which branch will be taken (T) or not (NT).
- Different schemes: static, dynamic, local, global,…
- Min and max search is very sensitive to branch prediction…
  ... though we can avoid this using CMOV instructions…

Computer Architecture: A Quantitative Approach (5th ed.), Hennessy & Patterson
Branch prediction

Branch predictors are used to avoid stalls on branches!

- Conditional instructions (such as the “if” statement) yield branches in the execution of a program.
- A misprediction can be quite expensive!
- The branch predictor will guess which branch will be taken (T) or not (NT).
- Different schemes: static, dynamic, local, global,…
- Min and max search is very sensitive to branch prediction...
  ... though we can avoid this using CMOV instructions...
  ... but still…

*Computer Architecture: A Quantitative Approach (5th ed.),* Hennessy & Patterson
Previous Work

- **Brodal & Moruz, 2005**: Mispredictions and (adaptive) sorting

---

**Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms**

Gerth Stølting Brodal and Gabriel Moruz

BRICS, Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark

{gerth, gabi}@daimi.au.dk

**Abstract.** Branch mispredictions is an important factor affecting the running time in practice. In this paper, we consider tradeoffs between the number of branch mispredictions and the number of comparisons for sorting algorithms by adaptive techniques. We present lower bounds on the number of branch mispredictions for deterministic comparison-based adaptive sorting algorithms for different measures of presortedness, given the upper bounds on the number of comparisons.
Previous Work

- **Brodal & Moruz, 2005**: mispredictions and (adaptive) sorting
- **Biggar et al., 2008**: experimental, branch prediction and sorting

An Experimental Study of Sorting and Branch Prediction
PAUL BIGGAR1, NICHOLAS NASH1, KEVIN WILLIAMS2 and DAVID GREGG
Trinity College Dublin

Sorting is one of the most important and well studied problems in Computer Science. Many good algorithms are known other factors. How architectures that s features, and while of general purpose properties. In this common sorting a) predictability of the b) mispredictions of a in a fashion which sort’s branches im effect on mergesort example the choice point out a simple and show also that predictability of its branch predictors a that two-level adapt Categories and Sub Systems Organizers

General Terms: Algorithms, Experimentation, Measurement

Additional Key Words and Phrases: Sorting, Branch Prediction

Permission to make digital/hard copy of all or part of this material for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the material, its date of publication, and the author(s) are prominently displayed, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

We refer to their associated branches as the i and j branches respectively.

ACM Journal Name, Vol. V, No. N, Month 20YY.

C. Nicaud

Fig. 8. (a) Shows of values of d. It multi-mergesort · per key for the al

Fig. 9. Overview of branch prediction behaviour in our quicksort implementations. Every figure
Previous Work

- Brodal & Moruz, 2005: mispredictions and (adaptive) sorting
- Biggar et al., 2008: experimental, branch prediction and sorting
- Sanders and Winkel, 2004: quicksort variant without branches

Super Scalar Sample Sort

Peter Sanders¹ and Sebastian Winkel²

¹ Max Planck Institut für Informatik
Saarbrücken, Germany, sanders@mpi-sb.mpg.de
² Chair for Prog. Lang. and Compiler Construction
Saarland University, Saarbrücken, Germany, sewi@cs.uni-sb.de

Abstract. Sample sort, a generalization of quicksort that partitions the input into many pieces, is known as the best practical comparison based sorting algorithm for distributed memory parallel computers. We show that

Fig. 2. Finding buckets using implicit search trees. The picture is for $k = 8$. We adopt the C convention that “$x > y$” is one if $x > y$ holds, and zero else.
Branch Misdectrions Don’t Affect Mergesort*  

Amr Elmasry1, Jyrki Katajainen1,2, and Max Stenmark2  

1 Department of Computer Science, University of Copenhagen  
   Universitetsparken 1, 2100 Copenhagen East, Denmark  
2 Jyrki Katajainen and Company  
   Thorsgade 101, 2200 Copenhagen North, Denmark  

Abstract. In quicksort, due to branch mispredictions, a skewed pivot-selection strategy can lead to a better performance than the exact-median pivot-selection strategy, or even sorting without branches. In this paper we investigate the behaviour of mergesort. By avoiding branches, we can avoid most mispredictions. When sorting a sequence of elements, mergesort performs $n \log_2 n + O(n)$ element comparisons and induces at most $O(n)$ branch mispredictions.

Table 3. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two in-situ variants of mergesort.

<table>
<thead>
<tr>
<th>Program</th>
<th>In-situ std::stable_sort</th>
<th>In-situ mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Branches Mispredicts</td>
</tr>
<tr>
<td>n</td>
<td>Per Ares</td>
<td></td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>49.2</td>
<td>29.7</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>57.6</td>
<td>35.0</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>62.7</td>
<td>38.5</td>
</tr>
<tr>
<td>$2^{25}$</td>
<td>68.0</td>
<td>41.3</td>
</tr>
</tbody>
</table>

1. test; 2. done = (q == t2); 3. if (done) goto exit; 4. entrance: 5. x = p; 6. a = p + 1; 7. y = q; 8. t = q + 1; 9. smaller = less(y, x); 10. if (smaller) a = t; 11. if (smaller) q = t; 12. if (! smaller) p = a; 13. if (! smaller) y = a; 14. x = x; 15. r = y; 16. --a; 17. ea = x; 18. +0+y; 19. done = (p == tl); 20. if (! done) goto test; 21. exit;
Previous Work

- Brodal & Moruz, 2005: mispredictions and (adaptive) sorting
- Biggar et al., 2008: experimental, branch prediction and sorting
- Sanders and Winkel, 2004: quicksort variant without branches
- Elmasry et al., 2012: mergesort variant without branches
- Kaligosi and Sanders, 2006: mispredictions and quicksort

---

**How Branch Mispredictions Affect Quicksort**

Kanela Kaligosi¹ and Peter Sanders²

¹ Max Planck Institute for
Saanbrueck
² Universität Karlsruhe
³ K. Kaligosi
⁴ P. Sanders

**Abstract.** We explain the counterintuitive observation that finding a “good” pivot close to the median does not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally “good” pivot (close to the median of the array to be partitioned) may not improve performance of quicksort.

**Table 1. Number of branch mispredictions**

<table>
<thead>
<tr>
<th></th>
<th>random pivot</th>
<th>α-skewed pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>static predictor</td>
<td>$\frac{3}{n} \lg n + O(n)$, $\frac{\ln 2}{n}$ ≈ 0.3466</td>
<td>$\frac{\ln 2}{n} \lg n + O(n)$, $\alpha &lt; 1/2$</td>
</tr>
<tr>
<td>1-bit predictor</td>
<td>$\frac{28 \ln 2}{45} n \lg n + O(n)$, $\frac{28 \ln 2}{45}$ ≈ 0.4621</td>
<td>$\frac{28 \ln 2}{45} n \lg n + O(n)$</td>
</tr>
<tr>
<td>2-bit predictor</td>
<td>$\frac{28 \ln 2}{45} n \lg n + O(n)$, $\frac{28 \ln 2}{45} \approx 0.4313$</td>
<td>$\frac{28 \ln 2}{45} n \lg n + O(n)$</td>
</tr>
</tbody>
</table>

**Fig. 3.** Time $/ n \lg n$ for random pivot, median of 3, exact median, 1/10-skewed pivot.
Previous Work

- **Brodal & Moruz, 2005**: mispredictions and (adaptive) sorting
- **Biggar et al., 2008**: experimental, branch prediction and sorting
- **Sanders and Winkel, 2004**: quicksort variant without branches
- **Elmasry et al., 2012**: mergesort variant without branches
- **Kalogi and Sanders, 2006**: mispredictions and quicksort
- **Martínez, Nebel and Wild, 2014**: mispredictions and quicksort

![Analysis of Branch Misses in Quicksort*](image)

**Abstract**

The analysis of algorithms mostly relies on counting classic elementary operations like additions, multiplications, comparisons, swaps etc. This approach is not efficient. Thus, processors and memory running time get a reliable sort: It has under certain conditions of a sample of a distribution degenerates to a different branch prediction (fat), 2-bit saturating counter (thin solid) and 2-bit flip-consecutive (dashed) using symmetric sampling: $t_{\text{CQS}} = (3l + 2, 3l + 2)$ and $t_{\text{YQS}} = (2l + 1, 2l + 1, 2l + 1)$.

Figure 5: Branch mispredictions, as a function of $t$, in CQS (black) and YQS (red) with 1-bit branch prediction (fat), 2-bit saturating counter (dashed) using symmetric sampling: $t_{\text{CQS}} = (3l + 2, 3l + 2)$ and $t_{\text{YQS}} = (2l + 1, 2l + 1, 2l + 1)$.

Figure 6: Branch mispredictions, as a function of $t$, in CQS (black) and YQS (red) with 1-bit (fat), 2-bit sc (thin solid) and 2-bit fc (dashed) predictors, using extremely skewed sampling: $t_{\text{CQS}} = (0, 6l + 4)$ and $t_{\text{YQS}} = (0, 6l + 3, 0)$.
Previous Work

- Brodal & Moruz, 2005: mispredictions and (adaptive) sorting
- Biggar et al., 2008: experimental, branch prediction and sorting
- Sanders and Winkel, 2004: quicksort variant without branches
- Elmasry et al., 2012: mergesort variant without branches
- Kaligosi and Sanders, 2006: mispredictions and quicksort
- Martínez, Nebel and Wild, 2014: mispredictions and quicksort
- Brodal and Moruz, 2006: skewed binary search trees

Skewed Binary Search Trees

Gerth Stølting Brodal\textsuperscript{1,*} and Gabriel Moruz\textsuperscript{2}

BRICS\textsuperscript{**}, Department of Computer Science, University of Aarhus, IT Parken,\nÅbogade 31, DK-8200 Århus N, Denmark. E-mail: \{gerald,gsait\}@imr.imi.au.dk

Abstract. It is well-known that a binary search tree should show a dominating factor between the number of cache faults and the layout of a binary search tree by several hundred percent. Branching to the left or right is equally costly, e.g., because of the self-balancing property of binary search trees. In this paper, we present a new class of skewed binary search trees that explicitly considers the ratio of the size of the tree. In this case, the tree is a fixed size. We show that in this case, the expected cost is \( O(1) \) and the cost for processing the right child is \( c_r = 0, 1, 2, \ldots, 28 \) (with \( c_r = 0 \) being the lowest curve).

Fig. 1. Bound on the expected cost for a random search, where the cost for visiting the left child is \( c_l = 1 \) and the cost for processing the right child is \( c_r = 0, 1, 2, \ldots, 28 \) (with \( c_r = 0 \) being the lowest curve).

Good predictions are worth... 5/16
Previous Work

- Brodal & Moruz, 2005: mispredictions and (adaptive) sorting
- Biggar et al., 2008: experimental, branch prediction and sorting
- Sanders and Winkel, 2004: quicksort variant without branches
- Elmasry et al., 2012: mergesort variant without branches
- Kaligosi and Sanders, 2006: mispredictions and quicksort
- Martínez, Nebel and Wild, 2014: mispredictions and quicksort
- Brodal and Moruz, 2006: skewed binary search trees

Skewed Binary Search Trees

Gerth Stølting Brodal\(^1\,*\) and Gabriel Moruz\(^2\)

BRICS\(^*\), Department of Computer Science, University of Aarhus, IT Parken, Åbogade 31, DK-8000 Århus N, Denmark. Email: \{gerth,ga\}@brics.dk

Abstract. It is well-known that a binary search tree should be skewed such that the number of cache faults per layout of a binary search tree by several hundred percent. Branching to the left or right is very same cost, e.g. because of the class of skewed bit binary search trees is fixed for this to work. In this paper, we present layouts of static skewed binary trees with a unit memory of the form $t = \frac{1}{2}$, $d = 0, 1, 2, \ldots, 28$ ($\alpha = 0$ being the lowest curve).

Good predictions are worth...
Proposition

*Expected number of mispredictions*, for the uniform distribution, on arrays of size $n$:

- **Naive Min Max Search:**
  \[ \sim 4 \log n \text{ for the 1-bit predictor} \]
  \[ \sim 2 \log n \text{ for the two 2-bit predictors and the 3-bit saturating counter.} \]

- **Optimized Min Max Search:**
  \[ \sim n/4 + O(\log n) \text{ for all four predictors.} \]

Idea of the proof:

- asymptotic analysis of the records in a random permutation,
- use the fundamental bijection that relates the records to the cycles in permutations,
- use classical results on the average number of cycles.
What if the distribution is not uniform?

**Definition (Ewens-like distribution for records)**

- To any $\sigma \in \mathfrak{S}_n$, we associate a weight $w(\sigma) = \theta^{\text{record}(\sigma)}$.
- Let $W_n = \sum_{\sigma \in \mathfrak{S}_n} w(\sigma) = \theta^{(n)}$ and $P(\sigma) = \frac{\theta^{\text{record}(\sigma)}}{\theta^{(n)}}$.

with $\theta^{(n)} = \theta(\theta + 1) \ldots (\theta + n - 1)$

Expected number of mispredictions:

- $\mu$: naive algorithm
- $\nu$: optimized algorithm

$\theta := \lambda n$.

$\mathbb{E}_n[\mu] \sim \mathbb{E}_n[\nu]$ for $\lambda_0 \approx 0.305$.

But optimized performs less comparisons, thus it becomes better before $\lambda_0$. 

$\frac{1}{n} \mathbb{E}_n[\mu]$  
$\frac{1}{n} \mathbb{E}_n[\nu]$
Exponentiation by squaring
Introducing unnecessary tests to speed up

POW(x,n)

```c
r = 1;
while (n > 0) {
    // n is odd
    if (n & 1)
        r = r * x;
    n /= 2;
    x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

\[ x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0} \]
Introducing unnecessary tests to speed up

POW(x,n)

```plaintext
r = 1;
while (n > 0) {
    // n is odd
    if (n & 1) \[P = \frac{1}{2}\]
        r = r * x;
    n /= 2;
    x = x * x;
}
```

\[x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}\]

x is a floating-point number, n is an integer and r is the result.
POW(x,n)

\[
\begin{align*}
    r &= 1; \\
    \text{while } (n > 0) \{ \\
        &\quad \text{// } n \text{ is odd} \\
        &\quad \text{if } (n & 1) P = \frac{1}{2} \\
        &\quad \quad r = r \cdot x; \\
        &\quad n /= 2; \\
        &\quad x = x \cdot x; \\
    \}
\end{align*}
\]

\(x\) is a floating-point number, \(n\) is an integer and \(r\) is the result.

\[x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}\]

UNROLLED(x,n)

\[
\begin{align*}
    r &= 1; \\
    \text{while } (n > 0) \{ \\
        &\quad t = x \cdot x; \quad \quad \text{// } n_0 == 1 \\
        &\quad \text{if } (n & 1) \\
        &\quad \quad r = r \cdot x; \\
        &\quad \quad n /= 2; \\
        &\quad \quad x = x \cdot x; \quad \quad \text{// } n_1 == 1 \\
        &\quad \text{if } (n & 2) \\
        &\quad \quad r = r \cdot t; \\
        &\quad \quad n /= 4; \\
        &\quad \quad x = t \cdot t; \\
    \}
\end{align*}
\]

\[x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}\]

25% more comparisons for guided than for unrolled exponential is 14% faster than the unrolled one; guided exponential is 29% faster than the classical one; yet, the number of multiplications is essentially the same.
Introducing unnecessary tests to speed up

POW(x,n)

\[
\begin{align*}
    r &= 1; \\
    \text{while} \ (n > 0) \ \{ \\
        &\quad \text{// } n \text{ is odd} \\
        &\quad \text{if} \ (n \ & 1) \ \mathbb{P} = \frac{1}{2} \\
        &\quad \quad r = r \times x; \\
        &\quad n /= 2; \\
        &\quad x = x \times x; \\
    \}\end{align*}
\]

x is a floating-point number, n is an integer and r is the result.

\[x^n = (x^2)^{\lfloor n/2 \rfloor} \cdot x^{n_0}\]

UNROLLED(x,n)

\[
\begin{align*}
    r &= 1; \\
    \text{while} \ (n > 0) \ \{ \\
        &\quad \quad t = x \times x; \\
        &\quad \quad \text{// } n_0 == 1 \\
        &\quad \quad \quad \text{if} \ (n \ & 1) \ \mathbb{P} = \frac{1}{2} \\
        &\quad \quad \quad \quad r = r \times x; \\
        &\quad \quad \quad \quad n /= 4; \\
        &\quad \quad \quad \quad x = t \times t; \\
    \}\end{align*}
\]

\[x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} \cdot x^{n_0}\]

25% more comparisons for UNROLLED than for POW.

Guided exponential is 14% faster than the unrolled one; guided exponential is 29% faster than the classical one; yet, the number of multiplications is essentially the same.
Introducing unnecessary tests to speed up

**POW(x,n)**

```plaintext
r = 1;
while (n > 0) {
    // n is odd
    if (n & 1) p = 1/2
        r = r * x;
    n /= 2;
    x = x * x;
}
```

**UNROLLED(x,n)**

```plaintext
r = 1;
while (n > 0) {
    t = x * x;
    // n0 == 1
    if (n & 1) p = 1/2
        r = r * x;
    if (n & 1) p = 1/2
        r = r * t;
    n /= 4;
    x = t * t;
}
```

**GUIDED(x,n)**

```plaintext
r = 1;
while (n > 0) {
    t = x * x;
    // n1n0! = 00
    if (n & 3){
        if (n & 1)
            r = r * x;
        if (n & 2)
            r = r * t;
    }
    n /= 4;
    x = t * t;
}
```

`x^n = (x^2)^{n/2} x^{n_0}`

`x^n = (x^4)^{n/4} (x^2)^{n_1} x^{n_0}`

x is a floating-point number, n is an integer and r is the result.
Introducing unnecessary tests to speed up

**POW(x, n)**

```plaintext
r = 1;
while (n > 0) {
    if (n & 1) \( P = \frac{1}{2} \)
        r = r * x;
    n /= 2;
    x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

\( x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0} \)

**UNROLLED(x, n)**

```plaintext
r = 1;
while (n > 0) {
    t = x * x;
    if (n & 1) \( P = \frac{1}{2} \)
        r = r * x;
    n /= 2;
    x = x * x;
}
```

\( x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0} \)

**GUIDED(x, n)**

```plaintext
r = 1;
while (n > 0) {
    t = x * x;
    if (n & 1) \( P = \frac{3}{4} \)
        if (n & 3) \( P = \frac{3}{4} \)
            r = r * x;
        else \( P = \frac{1}{2} \)
            if (n & 1) \( P = \frac{1}{2} \)
                r = r * t;
            n /= 4;
            x = t * t;
    }
}
```

25% more comparisons for **GUIDED** than for **UNROLLED** exponential is 14% faster than the **UNROLLED** one; **GUIDED** exponential is 29% faster than the classical one; yet, the number of multiplications is essentially the same.
Introducing unnecessary tests to speed up

POW(x,n)

```c
r = 1;
while (n > 0) {
    // n is odd
    if (n & 1) P = 1/2
    r = r * x;
    n /= 2;
    x = x * x;
}
```

x is a floating-point number, n is an integer and r is the result.

\[ x^n = (x^2)^{[n/2]} x^{n_0} \]

UNROLLED(x,n)

```c
r = 1;
while (n > 0) {
    t = x * x;
    // n_0 == 1
    if (n & 1) P = 1/2
    r = r * x;
    // n_1 == 1
    if (n & 2) P = 1/2
    r = r * t;
    n /= 4;
    x = t * t;
}
```

\[ x^n = (x^4)^{[n/4]} (x^2)^{n_1} x^{n_0} \]

GUIDED(x,n)

```c
r = 1;
while (n > 0) {
    t = x * x;
    // n_0 == 1
    if (n & 1) P = 1/2
    r = r * x;
    // n_1 == 1
    if (n & 2) P = 1/2
    r = r * t;
    n /= 4;
    x = t * t;
}
```

25% more comparisons for guided than for unrolled

guided exponential is 14% faster than the unrolled one;
guided exponential is 29% faster than the classical one;
yet, the number of multiplications is essentially the same.
Introducing unnecessary tests to speed up

\textbf{POW}(x,n)
\begin{verbatim}
r = 1;
while (n > 0) {
    // n is odd
    if (n & 1) P = \frac{1}{2}
    r = r \times x;
    n /= 2;
    x = x \times x;
}
\end{verbatim}

\textbf{UNROLLED}(x,n)
\begin{verbatim}
r = 1;
while (n > 0) {
    t = x \times x;
    // n₀ == 1
    if (n & 1) P = \frac{1}{2}
    r = r \times x;
    // n₁ == 1
    if (n & 2) P = \frac{1}{2}
    r = r \times t;
    n /= 4;
    x = t \times t;
}
\end{verbatim}

\textbf{GUIDED}(x,n)
\begin{verbatim}
r = 1;
while (n > 0) {
    t = x \times x;
    // n₀ == 1
    if (n & 1) P = \frac{3}{4}
    r = r \times x;
    // n₁n₀! = 00
    if (n & 3) P = \frac{3}{4}
    if (n & 1) P = \frac{2}{3}
    r = r \times t;
    if (n & 2) P = \frac{2}{3}
    r = r \times t;
    n /= 4;
    x = t \times t;
}
\end{verbatim}

\(x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}\)
\(x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}\)

- 25 \% more comparisons for GUIDED than for UNROLLED
Introducing unnecessary tests to speed up

POW(x,n)

\[ r = 1; \]
\[ \text{while } (n > 0) \{\]
\[ \quad \text{// } n \text{ is odd} \]
\[ \quad \text{if } (n \& 1) \; \mathbb{P} = \frac{1}{2} \]
\[ \quad r = r \ast x; \]
\[ \quad n /= 2; \]
\[ \quad x = x \ast x; \]
\[ \}\]

UNROLLED(x,n)

\[ r = 1; \]
\[ \text{while } (n > 0) \{\]
\[ \quad t = x \ast x; \]
\[ \quad \text{// } n_0 == 1 \]
\[ \quad \text{if } (n \& 1) \; \mathbb{P} = \frac{1}{2} \]
\[ \quad r = r \ast x; \]
\[ \quad \text{// } n_1 == 1 \]
\[ \quad \text{if } (n \& 2) \; \mathbb{P} = \frac{1}{2} \]
\[ \quad r = r \ast t; \]
\[ \quad n /= 4; \]
\[ \quad x = t \ast t; \]
\[ \}\]

GUIDED(x,n)

\[ r = 1; \]
\[ \text{while } (n > 0) \{\]
\[ \quad t = x \ast x; \]
\[ \quad \text{// } n_0 == 1 \]
\[ \quad \text{if } (n \& 3) \; \mathbb{P} = \frac{3}{4} \]
\[ \quad \text{if } (n \& 1) \; \mathbb{P} = \frac{2}{3} \]
\[ \quad r = r \ast x; \]
\[ \quad \text{// } n_1n_0! = 00 \]
\[ \quad \text{if } (n \& 2) \; \mathbb{P} = \frac{2}{3} \]
\[ \quad r = r \ast t; \]
\[ \quad n /= 4; \]
\[ \quad x = t \ast t; \]
\[ \}\]

\( x^n = (x^2)^{[n/2]} x^{n_0} \)

\( x^n = (x^4)^{[n/4]} (x^2)^{n_1} x^{n_0} \)

- 25\% more comparisons for GUIDED than for UNROLLED
- GUIDED exponential is 14\% faster than the UNROLLED one;
Introducing unnecessary tests to speed up

\[
\text{POW}(x,n) \begin{align*}
    r &= 1; \\
    \text{while} & \ (n > 0) \ {\{} \\
        & \ // \ n \ \text{is odd} \\
        & \ \text{if} \ (n \ & \ 1) \ \mathbb{P} = \frac{1}{2} \\
        & \ r = r \ * \ x; \\
        & \ n /= 2; \\
        & \ x = x \ * \ x; \\
    \} \\
\end{align*}
\]

\[x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0}\]

\[
\text{UNROLLED}(x,n) \begin{align*}
    r &= 1; \\
    \text{while} & \ (n > 0) \ {\{} \\
        & \ t = x \ * \ x; \\
        & \ // \ n_0 == 1 \\
        & \ \text{if} \ (n \ & \ 1) \ \mathbb{P} = \frac{1}{2} \\
        & \ r = r \ * \ x; \\
        & \ // \ n_1 == 1 \\
        & \ \text{if} \ (n \ & \ 2) \ \mathbb{P} = \frac{1}{2} \\
        & \ r = r \ * \ t; \\
        & \ n /= 4; \\
        & \ x = t \ * \ t; \\
    \} \\
\end{align*}
\]

\[x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}\]

\[
\text{GUIDED}(x,n) \begin{align*}
    r &= 1; \\
    \text{while} & \ (n > 0) \ {\{} \\
        & \ t = x \ * \ x; \\
        & \ // \ n_0 == 1 \\
        & \ \text{if} \ (n \ & \ 1) \ \mathbb{P} = \frac{1}{2} \\
        & \ r = r \ * \ x; \\
        & \ // \ n_1 == 00 \\
        & \ \text{if} \ (n \ & \ 3) \ {\{} \ \mathbb{P} = \frac{3}{4} \\
        & \ \text{if} \ (n \ & \ 1) \ \mathbb{P} = \frac{2}{3} \\
        & \ r = r \ * \ x; \\
        & \ \text{if} \ (n \ & \ 2) \ \mathbb{P} = \frac{2}{3} \\
        & \ r = r \ * \ t; \\
        & \ n /= 4; \\
        & \ x = t \ * \ t; \\
    \} \\
\end{align*}
\]

- 25 % more comparisons for GUIDED than for UNROLLED
- GUIDED exponential is 14% faster than the UNROLLED one;
- GUIDED exponential is 29% faster than the classical one;
Introducing unnecessary tests to speed up

### POW(x,n)
```c
r = 1;
while (n > 0) {
    // n is odd
    if (n & 1) P = \frac{1}{2}
    r = r * x;
    n /= 2;
    x = x * x;
}
```

- `x` is a floating-point number, `n` is an integer and `r` is the result.

- \[ x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0} \]

### UNROLLED(x,n)
```c
r = 1;
while (n > 0) {
    t = x * x;
    // n_0 == 1
    if (n & 1) P = \frac{1}{2}
    r = r * x;
    // n_1 == 1
    if (n & 2) P = \frac{1}{2}
    r = r * t;
    n /= 4;
    x = t * t;
}
```

- \[ x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0} \]

### GUIDED(x,n)
```c
r = 1;
while (n > 0) {
    t = x * x;
    // n_0 == 1
    if (n & 1) P = \frac{1}{2}
    r = r * x;
    // n_1 == 00
    if (n & 3){ P = \frac{3}{4}
        if (n & 1) P = \frac{2}{3}
            r = r * x;
        if (n & 2) P = \frac{2}{3}
            r = r * t;
    }
    n /= 4;
    x = t * t;
}
```

- 25% more comparisons for GUIDED than for UNROLLED
- GUIDED exponential is 14% faster than the UNROLLED one;
- GUIDED exponential is 29% faster than the classical one;
- yet, the number of multiplications is essentially the same.
**Guided Pow: average number of mispredictions**

### Theorem

**Compute** $x^n$, for random $n$ in $\{0, \ldots, N-1\}$.

- **Expected nb. of conditionals:**
  \[ \sim \log_2 N \text{ for classical and unrolled pow} \]
  \[ \sim \frac{5}{4} \log_2 N \text{ for the guided one} \]

- **Expected nb. of mispredictions:**
  \[ \sim \frac{1}{2} \log_2 N \text{ for classical and unrolled pow} \]
  \[ \sim \left( \frac{1}{2} \mu\left(\frac{3}{4}\right) + \frac{3}{4} \mu\left(\frac{2}{3}\right) \right) \log_2 N \text{ for guided pow} \]

```c
r = 1;
while (n > 0) {
    t = x * x;
    // n_1n_0! = 00
    if (n & 3) {
        if (n & 1)
            r = r * x;
        if (n & 2)
            r = r * t;
    }
    n /= 4;
    x = t * t;
}
```

Number of mispredictions (Ergodic Th.):

\[ \mathbb{E}[M_n] \sim \mathbb{E}[L_n] \times \mu(p) \]

$L_n$: length of the path in the Markov chain, and

\[ \mu(p) = \sum_{(i,j) \in \text{mispred}} \pi_p(i) \cdot M_p(i,j). \]

\[ \mu\left(\frac{3}{4}\right) = \frac{3}{10} \text{ and } \mu\left(\frac{2}{3}\right) = \frac{2}{5} \]
Guided Pow: average number of mispredictions

**Theorem**

Compute $x^n$, for random $n$ in $\{0, \ldots, N - 1\}$.

- **Expected nb. of conditionals:**
  - $\sim \log_2 N$ for classical and unrolled pow
  - $\sim \frac{5}{4} \log_2 N$ for the guided one

- **Expected nb. of mispredictions:**
  - $\sim \frac{1}{2} \log_2 N$ for classical and unrolled pow
  - $\sim 0.45 \log_2 N$ for guided pow (2-bit pred.)

```
 Guided(x,n)
 r = 1;
 while (n > 0) {
   t = x * x;
   // n1n0! = 00
   if (n & 3) {
     if (n & 1)
       r = r * x;
     if (n & 2)
       r = r * t;
   }
   n /= 4;
   x = t * t;
 }
```

$\mu(\frac{3}{4}) = \frac{3}{10}$ and $\mu(\frac{2}{3}) = \frac{2}{5}$

- 25% more comparisons than unrolled
- unnecessary if: added mispred.
- other ones: less mispred.

$\triangleright$ 5% less mispred. (2-bit predictor)
$\triangleright$ 11% less mispred. (3-bit predictor)

N. Auger, C. Nicaud, C. Pivoteau

Good predictions are worth... 10/16
Good Predictions Are Worth a Few Comparisons

As expected at this point in our work, the changes we brought in the binary search are quite sensitive to the division by powers of two, which are simple binary shifts, as in the initial binary search (see Figure 7).

5.1 Unbalancing the Binary Search

Carrying on with the divide and conquer strategy but partitioning the array into three parts is taken with probability \( \frac{1}{3} \). A simple way to change that is to partition another way, for instance with parts of size about \( \frac{n}{3} \) and \( \frac{2n}{3} \). This predictor is not efficient enough to offset the mispredictions caused by the additional conditional branch that is searched for to the middle of the array.

5.2 Experiments

To limit the cost of partitioning, we consider arrays of uniform random floating-point numbers, we get a conditional branch that is accessed. Thus we conducted experiments on arrays that fit in the last-level cache of our machine.

UnrolledPow

\[
m = \frac{(d+f)}{2};
\]

\[
\text{if } (T[m] < x) \{ \text{BiasedBinarySearch}(m1, x, n); \}
\]

\[
\text{else if } (m < 0) \{ \text{BiasedBinarySearch}(m1, x, n); \}
\]

\[
\text{else} \{ \text{BiasedBinarySearch}(m1, x, n); \}
\]

GuidedPow

\[
m = \frac{d}{4} + \frac{f}{4};
\]

\[
\text{if } (T[m] < x) \{ \text{SkewSearch}(m1, x, n); \}
\]

\[
\text{else if } (m < 0) \{ \text{SkewSearch}(m1, x, n); \}
\]

\[
\text{else} \{ \text{SkewSearch}(m1, x, n); \}
\]
5.1 Unbalancing the Binary Search

We first consider the classical binary search which partitions a sorted array of size $n$ into two equal parts $n/2$ and $n/2$. This predictor is not better than the classical binary search and the number of comparisons than $\log_2 n$.

Carrying on with the divide and conquer strategy but partitioning the array into three parts $n/4$, $3n/4$, and $n/2$, gives a ternary search. The main issue with this approach is that, in practice, the number of mispredictions is greater for $\log_3 n$ than for $\log_2 n$.

Using Theorem 3 and Equations (1) and (2), we get that

$$f = \frac{\log_2 n}{\log_3 n}$$

is the number that is searched for.

5.2 Experiments

We conducted experiments on arrays of uniform random floating-point numbers, we get a conditional branch that is accessed. Thus we conducted experiments on arrays that fit in the last-level cache of our processor, since the way we partition the array influences the location where the memory is accessed. Therefore, we experimentally performed

$$f = \frac{\log_2 n}{\log_3 n}$$

for the 1-bit, 2-bit saturated, flip-on-consecutive, as in the initial binary predictions, since the way we partition the array influences the location where the memory is accessed.

UnbalancedSearch

$$\text{BiasedBinarySearch}$$

$\text{BinarySearch}$

$\text{UnrolledPow}$

$\text{GuidedPow}$

$\text{BiasedBinarySearch}$

$\text{SkewSearch}$
Unbalancing the binary search

**BinarySearch**

- \( n/2 \) \( n/2 \)

**BiasedBinarySearch**

- \( n/4 \) \( 3n/4 \)

**SkewSearch**

- \( n/4 \) \( n/4 \) \( n/2 \)

5.1 Unbalancing the Binary Search

In both cases, \( n \) is accessed. Thus we conducted experiments on arrays that fit in the last-level cache of our machine, since the way we partition the array influences the location where the memory is accessed. Therefore, to limit the cost of partitioning, we choose to slice the array into two parts of size \( n/2 \), as in the initial binary search. This predictor is not enough to obtain a good performance, especially if the access pattern is more or less uniform. A simple way to change that is to partition another way, for example, partitioning the array into three parts of size \( n/4 \) and \( 3n/4 \), which gives a ternary search. The main issue with this approach is that, in practice, the ternary search (see Figure 7) experimentally performs much better. Unlike the classical binary search and the BiasedBinarySearch, it performs better than the BiasedBinarySearch of the other two algorithms. In particular, for the 1-bit predictor, the expected number of comparisons is \( \frac{n}{2} + \frac{n}{4} \), and for the 3-bit predictor, it is \( \frac{n}{4} + \frac{3n}{4} \). This can also be done by setting the mispredictions caused by the additional conditional. For the 3-bit saturated counter, \( T[m1] = \frac{n}{4} + \frac{3n}{4} \), but for the 1-bit predictor, \( T[m1] = \frac{n}{2} + \frac{n}{4} \) if \( x \) is equal to \( n \), \( 1 \), and \( 0 \), or \( n \), \( 1 \), and \( 0 \), or \( n \), \( 1 \), and \( 0 \). The expected number of comparisons is therefore equal to \( 11 \log_2 n \). This is more than for the 1-bit, 2-bit saturated, flip-on-consecutive, and 11 log 2 n predictor, but it is equal to \( 11 \log_2 n \). It is therefore used in our previous examples, the changes we brought in the binary search are quite sensitive to the division by powers of two, which are simple binary shifts, as in the initial binary search (see Figure 7).
Unbalancing the binary search

![Graph showing time vs. array size for different search methods on an Intel Core i7]

- **binary search**
- **biased binary search**
- **skew search**

Time (in nsec.) vs. Array size
Analysis of the local predictor

Theorem

For arrays of size $n$ filled with random uniform integers. $C_n$ is the number of comparisons and $M_n$ the number of mispredictions.

$$
E[C_n] = \frac{\log n}{\log 2} \quad \text{BiasedBinarySearch} \quad \frac{4\log n}{4\log 4 - 3\log 3} \quad \text{SkewSearch} \quad \frac{7\log n}{6\log 2}
$$

$$
E[M_n] = \frac{\log n}{2\log 2} \quad \mu(\frac{1}{4})E[C_n] \quad \left(\frac{4}{7}\mu(\frac{1}{4}) + \frac{3}{7}\mu(\frac{1}{3})\right)E[C_n]
$$

$\mu$ is the expected misprediction probability associated with the predictor.

Idea of the proof:

- Get the expected number of times a given conditional is executed by Roura’s Master Theorem [Rou01].
- Ensure that our predictors behave almost like Markov chains.
Analysis of the local predictor

For arrays of size $n$ filled with random uniform integers, $C_n$ is the number of comparisons and $M_n$ the number of mispredictions.

<table>
<thead>
<tr>
<th></th>
<th>BinarySearch</th>
<th>BiasedBinarySearch</th>
<th>SkewSearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E[C_n]$</td>
<td>$1.44 \log n$</td>
<td>$1.78 \log n$</td>
<td>$1.68 \log n$</td>
</tr>
<tr>
<td>$E[M_n]$</td>
<td>$0.72 \log n$</td>
<td>$0.53 \log n$</td>
<td>$0.58 \log n$</td>
</tr>
</tbody>
</table>

with a 2-bit saturated counter.

Idea of the proof:

- Get the expected number of times a given conditional is executed by Roura’s Master Theorem [Rou01].
- Ensure that our predictors behave *almost* like Markov chains.
Almost like Markov chains?

Expected number of iterations $L(n)$ of \textsc{BiasedBinarySearch}:

$$L(n) = 1 + \frac{a_n}{n+1} L(a_n) + \frac{b_n}{n+1} L(b_n),$$

with $a_n = \left\lfloor \frac{n}{4} \right\rfloor + 1$, $b_n = \left\lceil \frac{3n}{4} \right\rceil$

and $L(0) = 0$

But $\frac{a_n}{n+1}$ and $\frac{b_n}{n+1}$ are not fixed anymore...

\begin{center}
\begin{tikzpicture}
  \node (0) at (0,0) {0,8};
  \node (1) at (-3,-3) {0,2};
  \node (2) at (-6,-6) {0,0};
  \node (3) at (-4,-3) {1,2};
  \node (4) at (-7,-6) {1,1};
  \node (5) at (-2,-3) {3,8};
  \node (6) at (-5,-6) {3,4};
  \node (7) at (-3,-6) {3,3};
  \node (8) at (-1,-3) {5,8};
  \node (9) at (-4,-6) {4,4};
  \node (10) at (-2,-6) {3,3};
  \draw (0) -- (1) node [midway, above] {$\frac{1}{3}$};
  \draw (0) -- (5) node [midway, above] {$\frac{2}{3}$};
  \draw (1) -- (2) node [midway, above] {$\frac{1}{3}$};
  \draw (1) -- (3) node [midway, above] {$\frac{2}{3}$};
  \draw (2) -- (4) node [midway, above] {$\frac{1}{2}$};
  \draw (2) -- (4) node [midway, above] {$\frac{1}{2}$};
  \draw (3) -- (6) node [midway, above] {$\frac{1}{3}$};
  \draw (3) -- (6) node [midway, above] {$\frac{1}{3}$};
  \draw (4) -- (7) node [midway, above] {$\frac{1}{2}$};
  \draw (4) -- (7) node [midway, above] {$\frac{1}{2}$};
  \draw (6) -- (9) node [midway, above] {$\frac{1}{4}$};
  \draw (6) -- (9) node [midway, above] {$\frac{1}{4}$};
  \draw (8) -- (10) node [midway, above] {$\frac{3}{4}$};
\end{tikzpicture}
\end{center}

The trick...

The probability that the path $\mathcal{P}$ taken by \textsc{BiasedBinarySearch} in the decomposition tree differs from the one taken in the ideal tree at one of the first $\text{length}(\mathcal{P}) - \sqrt{\log n}$ steps is $O(\frac{1}{\log n})$.

Good predictions are worth...
What about a global predictor?

```c
1. d = 0; f = n;
2. while (d < f){
3.     m1 = (3*d+f)/4;
4.     if (T[m1] > x) f = m1;
5.         else {
6.             m2 = (d+f)/2;
7.             if (T[m2] > x){
8.                 f = m2;
9.                 d = m1+1;
10.             }
11.         }
12.     else d = m2+1;
13. }
14. return f;
```

Good predictions are worth...
Gerth Stølting Brodal and Gabriel Moruz.
Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms.

Gerth Stølting Brodal and Gabriel Moruz.
Skewed Binary Search Trees.

Paul Biggar, Nicholas Nash, Kevin Williams, and David Gregg.
An experimental study of sorting and branch prediction.

Amr Elmasry, Jyrki Katajainen, and Max Stenmark.
Branch Mispredictions Dont Affect Mergesort.

John L. Hennessy and David A. Patterson.

Kanela Kaligosi and Peter Sanders.
How Branch Mispredictions Affect Quicksort.

Conrado Martínez, Markus E. Nebel, and Sebastian Wild.
Analysis of branch misses in quicksort.

Salvador Roura.
Improved master theorems for divide-and-conquer recurrences.