Lean programs, branch mispredictions, and sorting

Amr Elmasry & Jyrki Katajainen

Department of Computer Science
University of Copenhagen

These slides are available at http://www.cphstl.dk

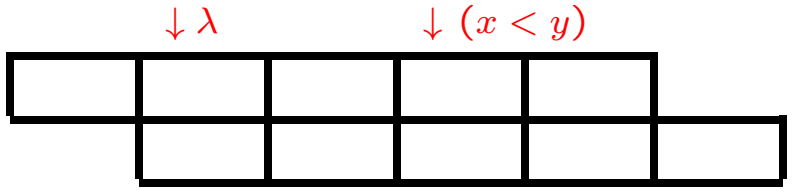
Problem: Pipelining

Code

if $(x < y)$ **goto** λ;
    I_1;
    I_2;
    ...
λ:
    J_1;
    J_2;
    ...

Pipelined execution

if $(x < y)$ **goto** λ;
I_1 or J_1**?**



Here instructions are carried out in five steps:

- Instruction fetch
- Register read
- Execution
- Data access
- Register write

History table $\rightarrow$ prediction $\rightarrow$ speculation $\overset{\text{if wrong}}{\rightarrow}$ cycles wasted

Early work

Call for research: "the frequency of conditional jump instructions might also be a factor"

[Knuth 1993; The Stanford GraphBase, p. 497]

Mergesort: $O(n \lg n)$ work, $n \lg n + O(n)$ element comparisons, and $O(n)$ branch mispredictions, where n is the number of elements being sorted; the stronger claims made in the thesis are wrong.

[Mortensen 2001; Master's Thesis]

Main idea

Decouple element comparisons from conditional branches!

C++ code

```
1   if (less(*q, *p)) {
2     *r = *q;
3     ++q;
4   }
5   else {
6     *r = *p;
7     ++p;
8   }
9   ++r;
```

Assembly-language code

```
 1  movl   (%eax), %edx
 2  leal   4(%eax), %edi
 3  movl   (%ebx), %ecx
 4  leal   4(%ebx), %ebp
 5  cmpl   %ecx, %edx
 6  cmovge %ecx, %edx
 7  cmovge %ebp, %ebx
 8  cmovl  %edi, %eax
 9  movl   %edx, (%esi)
10  addl   $4, %esi
```

Aha! Conditional move `if (c) x = y`

Later work—confuses me

Samplesort: $O(n \lg n)$ work, $n \lg n + O(n)$ element comparisons, and $O(n)$ branch mispredictions **on an average**

[Sanders & Winkel 2004]

Lower bound: Branch mispredictions are unavoidable in sorting

[Brodal & Moruz 2005]

Quicksort: A skewed pivot-selection strategy can lead to a better performance than the exact-median pivot-selection strategy

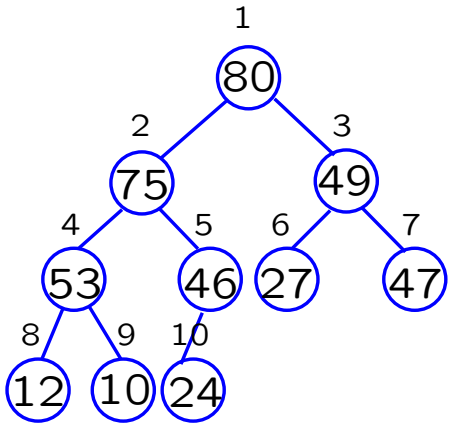
[Kaligosi & Sanders 2006]

Search trees: Skewed binary search trees can perform better than perfectly balanced search trees

[Brodal & Moruz 2006]

Appetizer: Heap construction

```
 1 template <typename position, typename index, typename comparator>
 2 void siftdown(position a, index i, index n, comparator less) {
 3   typedef typename std::iterator_traits<position>::value_type element;
 4   element copy = a[i];
 5 loop:
 6   index j = 2 * i;
 7   if (j <= n) {
 8     if (j < n)
 9       if (less(a[j], a[j + 1]))
10         j = j + 1;
11     if (less(copy, a[j])) {
12       a[i] = a[j];
13       i = j;
14       goto loop;
15     }
16   }
17   a[i] = copy;
18 }
19
20 template <typename position, typename comparator>
21 void make_heap(position first, position beyond, comparator less) {
22   typedef typename std::iterator_traits<position>::difference_type index;
23   position const a = first - 1;
24   index const n = beyond - first;
25   for (index i = n / 2; i > 0; --i)
26     siftdown(a, i, n, less);
27 }
```



[Floyd 1964]

Optimization 1

opt$_1$: Make sure that `siftdown` is always called with an odd `n`.

```
> template <typename position, typename index, typename comparator>
> void siftup(position a, index j, comparator less) {
>     ...
> }
```

```
>   index const m = (n & 1) ? n : n - 1;
>   for (index i = m / 2; i > 0; --i)
>     siftdown(a, i, m, less);
>   siftup(a, n, less);
```

```
 8     if (j < n)

25   for (index i = n / 2; i > 0; --i)
26     siftdown(a, i, n, less);
```

Execution time/n [ns]

Program / n	F	F_1
2^{10}	11.4	10.3
2^{15}	11.4	10.5
2^{20}	16.2	16.1
2^{25}	16.4	15.6

Aha! An unnecessary if

Aha! Cache effects

Optimization 2

opt$_2$: Interpret the result of a comparison as an integer and use this value in normal index arithmetic.

```
>      j = j + less(a[j], a[j + 1]);
```

```
 9      if (less(a[j], a[j + 1]))
10        j = j + 1;
```

Execution time/n [ns]

Program / n	F_1	F_{12}
2^{10}	10.3	7.1
2^{15}	10.5	7.6
2^{20}	16.1	11.0
2^{25}	15.6	14.0

Aha! A mixture of `int`s and `bool`s

Optimization 3

opt$_3$: Do not make any element moves when the element at the root stays in its original location.

```
>   element copy;
>   index k = 2 * i;
>   k = k + less(a[k], a[k + 1]);
>   if (less(a[i], a[k])) {
>     copy = a[i];
>     a[i] = a[k];
>   }
>   else {
>     return;
>   }
>   i = k;

4   element copy = a[i];
```

Aha! Loop unrolling

Execution time/n [ns]

Program / n	F$_{12}$	F$_{123}$
2^{10}	7.1	6.4
2^{15}	7.6	6.8
2^{20}	11.0	10.0
2^{25}	14.0	12.9

Element moves/n

Program / n	F	F$_{123}$
2^{10}	1.73	1.52
2^{15}	1.74	1.53
2^{20}	1.74	1.53
2^{25}	1.74	1.52

Ultimate goal: Lean programs

Referee comment: Conditional-branch-lean would be a better term!

- A program that has a **constant** number of unnested loops.
- Each loop is **branch-free**, except the final conditional branch at the end.
- A branch predictor is **static** assuming that forward branches are not taken and backward branches are taken.
- Each such program induces $O(1)$ branch mispredictions in this model.

Our main result: Program transformation

Theorem. Let $\mathcal{P}$ be a program of length κ, measured in the number of assembly-language instructions. Assume that the running time of $\mathcal{P}$ is $t(n)$ for an input of size n. There exists a program $\mathcal{Q}$ of length $O(\kappa)$ that is equivalent to $\mathcal{P}$, runs in $O(\kappa t(n))$ time for the same input as $\mathcal{P}$, and induces $O(1)$ branch mispredictions.

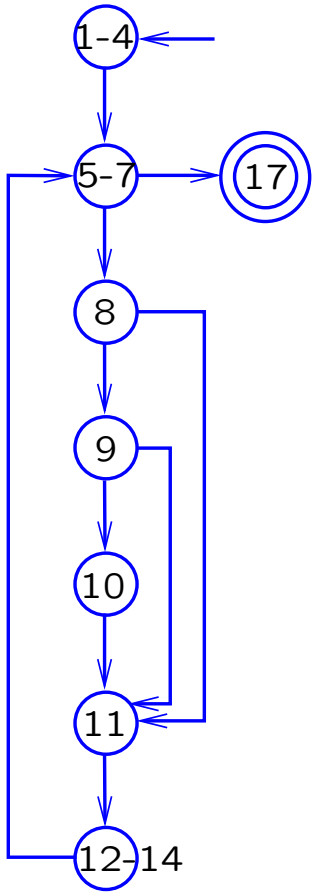
[this paper]

An improvement

Referee comment: It seems that the bound on the running time could be improved.

Yes. Instead of program length κ, one could express the running time as a function of the number of basic blocks. A **basic block** is a piece of code with at most one branch or branch target; branch targets start a block and branches end a block.

Example: The control-flow graph of `siftdown`



Other results: Hand-tailoring

Heapsort: $O(n \lg n)$ work, $2n \lg n + O(n)$ element comparisons, $O(1)$ extra space, and $O(1)$ branch mispredictions

[this paper]

Mergesort: $O(n \lg n)$ work, $n \lg n + O(n)$ element comparisons, $O(n)$ extra space, and $O(1)$ branch mispredictions

[this paper]

In-situ mergesort: $O(n \lg n)$ work, $n \lg n + O(n)$ element comparisons, $O(\lg n)$ extra space, and $O(n)$ branch mispredictions

[Elmasry, Katajainen & Stenmark 2012]

Criticism

Theory

1) We used C++ to describe the programs.
2) We relied on conditional-move instructions.
3) We assumed that the branch predictor was static.
4) On paper everything worked smoothly.

Practice

1) Assembly code written by us was much slower than that generated by the compiler.
2) We could not force the compiler to translate them as we wanted.
3) Real branch-prediction hardware is more complicated.
4) We got test results that we could not explain.

Advice for practitioners

- Write programs as before if speed is not primary concern.
- Keep easy-to-predict branches since they have small overhead on modern processors.
- Eliminate hard-to-predict branches if the elimination will not cause too much overhead.

Concluding remarks

- Welcome to the world of paranoid programming!

Referee comment: How architecture-dependent are the results?

Referee comment: The fun factor is pretty much non-existent.

- It was fun to tailor the programs until we saw the pattern how to write them.
- Still, we do not know what is the most efficient way of avoiding `if` statements.

Aha! Creativity still needed