Quicksort Improvements - 58 Years Later

Konrad Rafał Witaszczyk
nlh930@alumni.ku.dk
def@FreeBSD.org

University of Copenhagen

November 7, 2019
This presentation is mostly based on the project report:

Quicksort Improvements - 57 Years Later

By Konrad Rafał Witaszczyk and Pavel Kucera
Quicksort

QUICKSORT(A, p, r)
   if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q - 1)
      QUICKSORT(A, q + 1, r)

Source: CLRS, Introduction to algorithms.
Quicksort

QUICKSORT(A, p, r)
if p < r
q = PARTITION(A, p, r)
QUICKSORT(A, p, q - 1)
QUICKSORT(A, q + 1, r)

Source: CLRS, Introduction to algorithms.

▶ Worst-case running time:
Quicksort

QUICKSORT(A, p, r) 
if p < r 
    q = PARTITION(A, p, r) 
    QUICKSORT(A, p, q - 1) 
    QUICKSORT(A, q + 1, r) 

Source: CLRS, Introduction to algorithms.

▶ Worst-case running time: $O(n^2)$;
Quicksort

QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)

Source: CLRS, Introduction to algorithms.

- Worst-case running time: $O(n^2)$;
- Expected running time:
Quicksort

QUICKSORT(A, p, r)
if p < r
  q = PARTITION(A, p, r)
  QUICKSORT(A, p, q - 1)
  QUICKSORT(A, q + 1, r)

Source: CLRS, Introduction to algorithms.

- Worst-case running time: $O(n^2)$;
- Expected running time: $O(n \log n)$;
Quicksort

**QUICKSORT** *(A, p, r)*

if *p* < *r*

q = **PARTITION** *(A, p, r)*

QUICKSORT *(A, p, q - 1)*

QUICKSORT *(A, q + 1, r)*

Source: CLRS, Introduction to algorithms.

- **Worst-case running time:** \(O(n^2)\);
- **Expected running time:** \(O(n \log n)\);
- **Constant in** \(O(n \log n)\) **is quite small;**
Quicksort

```c
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

Source: CLRS, Introduction to algorithms.

- Worst-case running time: $O(n^2)$;
- Expected running time: $O(n \log n)$;
- Constant in $O(n \log n)$ is quite small;
- It’s an in-place algorithm;
QuickSort

QUICKSORT(A, p, r)
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)

Source: CLRS, Introduction to algorithms.

- Worst-case running time: $O(n^2)$;
- Expected running time: $O(n\log n)$;
- Constant in $O(n\log n)$ is quite small;
- It's an in-place algorithm;
- It's an unstable algorithm.
Motivation

- Inspired by the paper 'Heap Construction – 50 Years Later', Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen;
- There are many algorithms that improve the original Quicksort algorithm;
- Improving an algorithm for one property can introduce additional penalty in other metrics, for example branch mispredictions and number of element moves;
- We’d like to verify statements from scientific papers about proposed algorithms;
- We’d like to measure number of comparisons, moves, cache misses and branch mispredictions and see if an algorithm is optimal for all these metrics;
- Is there any framework to measure performance in these terms that would be accurate, portable and easy to use?
Selected algorithms

1. Hoare’s Quicksort.
2. Tuned Quicksort.
   Median-of-three pivot selection.
3. Instrosort.
   Quicksort with heapsort after $2 \times \log n$ recursion depth.
4. Skewed Introsort.
   Introsort with a skewed pivot.
5. Super Scalar Samplesort.
   Faster than `std::sort` in many cases but uses 2-3x additional memory.
   Cache-efficient, avoids branch-mispredictions.
7. Standard C++ library.
Goal

What is a running time, a number of comparisons, element moves, branch mispredictions and cache misses of each algorithm?
Performance tools

During the project we tried the following tools and techniques:

- DTrace;
- pmcstat;
- Processor Counter Monitor (PCM);
- Instruments (Xray);
- Instrumentation;
- Simulation.
Accuracy vs real case metrics

In order to measure resource usage in parts of any program we must refer to them using assembly symbols.

However, in order to run a program as it was used in a production environment we must use compiler optimizations that remove assembly symbols. In this case we can still measure performance relative to some baseline and hope that it was enough accurate.
Name mangling

Encoding function prototypes into unique names.

It is used by a compiler (for example for overloading) and a linker. Each compiler implements its own name mangling algorithm, e.g. for GCC 8.2.0 we have:

```c
void hoare::sort<int *, std::less<int> >(int *, int *, std::less<int>);
_ZN5hoare4sort1PiSt4lessliEEEvT_S4_T0_
```
Name mangling: utils/demangle and utils/findsymbol

Using GCC ABI (abi::__cxa_demangle), `demangle` prints a function prototype for a given symbol:

```
$ ./utils/demangle _ZN5hoare4sortIPiSt4lessliEEEvT_S4_T0_
void hoare::sort<int *, std::less<int> >(int *, int *, std::less<int> )
$
```

Using `nm` and `demangle`, `findsymbol` prints all symbols corresponding to function prototypes matching a regular expression in a binary file:

```
$ ./utils/findsymbol.sh ./hoare 'Element<int>::operator=(Element<int> const&)
_ZN7ElementLiEaSERKS0_
$
```
DTrace: running time (benchmark/time.d)

```d
uint64_t total;
self uint64_t depth, start;

BEGIN
{
    self->depth = 0;
    total = 0;
}

pid$target::$1:entry
/ self->depth == 0 /
{
    self->start = vtimestamp;
}

pid$target::$1:entry
{
    self->depth = self->depth + 1;
}

pid$target::$1:return
{
    self->depth = self->depth - 1;
}

pid$target::$1:return
/ self->depth == 0 /
{
    total = (total + vtimestamp - self->start);
}

END
{
    printf("%u", total / 1000);
}
```
DTrace: running time (benchmark/time.d)

For hoare::sort we can execute:

```bash
$ sudo dtrace -s benchmark/time.d -c './hoare input.txt' \ 
  _ZN5hoare4sortIPiSt4lessliEEEvT_S4_T0_ 82695 $ 
```
DTrace: number of moves (benchmark/count.d)

```d
uint64_t ncalls;

BEGIN
{
    ncalls = 0;
}

pid$target::$1:entry
{
    ncalls = ncalls + 1;
}

END
{
    printf("%lu", ncalls);
}
```
DTrace: number of moves (benchmark/count.d)

For `Element<int>::operator=(Element<int> const&)` we can execute:

```bash
$ sudo dtrace -s benchmark/count.d -c './hoare input.txt' ← _ZN7ElementliEaSERKS0_
90897
$ 
```
DTrace for Solaris includes cpc provider that implements probes for CPU performance counters. Unfortunately, the provider has not been ported to FreeBSD, macOS or Linux.
pmcstat

pmcstat is a performance measurement utility on FreeBSD that gives access to CPU counters via hwpmc driver, including:

```bash
# pmccontrol -L
...
   BR_INST RETIRIED . ALL BRANCHES
...
   MEM_LOAD_UOPS RETIRED . L1 MISS
   MEM_LOAD_UOPS RETIRED . L2 MISS
...
```
We use Intel Core i7-3610QM CPU. In Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3 we find the meaning of the counters:

- **BR_MISP RETIRED . ALL_BRANCHES** – mispredicted branch instructions at retirement;
- **MEM_LOAD_UOPS RETIRED . L1_MISS** – retired load uops whose data source followed an L1 miss;
- **MEM_LOAD_UOPS RETIRED . L2_MISS** – retired load uops that missed L2, excluding unknown sources.
We run `pmcstat` in counting and sampling modes for a user process and later calculate results:

```bash
$ pmcstat -S BR_MISP RETIRED.ALL_BRANCHES -P BR_MISP RETIRED.ALL_BRANCHES \n   -O hoare.pmcstat ./hoare input.txt
$ pmcstat -R hoare.pmcstat -G

07.08% [1069] _ZN5hoareL9partitionIPiSt4lessliEEET_S4_S4_T0_ @ /ztank/priv/KU/AE/project/src/hoare

According to `pmcstat` there were 1069 branch mispredictions in the partition function of the original Quicksort implementation. It's 7.08% of all branch mispredictions that occurred in the program.
As mentioned before, explained methods does not allow us to use compiler optimizations. We decided to introduce code instrumentation and use the -03 optimization level.
We introduce the following counters:

```c
#define MEASURE_COMPARISONS
static uint64_t ncomparisons = 0;
#endif

#define MEASURE_MOVES
static uint64_t nmoves = 0;
#endif

#define MEASURE_TIME
static struct timespec dtime;
#endif
```
Instrumentation

In case of measuring comparisons or moves we wrap elements by a class `Element` in which we define:

```cpp
#ifdef MEASURE_MOVES
    Element(Element<T> const &element) {
        *this = element;
    }

    Element& operator=(Element const &element) {
        this->value = element.value;
        nmoves++;
        return (*this);
    }
#endif

friend bool operator<(const Element<T> &x, const Element<T> &y) {
    #ifdef MEASURE_CONTRASTS
        ncomparisons++;
    #endif
    return (x.value < y.value);
}

friend bool operator==(const Element<T> &x, const Element<T> &y) {
    #ifdef MEASURE_CONTRASTS
        ncomparisons++;
    #endif
    return (x.value == y.value);
}
```
Instrumentation

In case of measuring running time we calculate a difference between the time after calling and the time before calling sort:

```c
#define MEASURE_TIME
    clock_gettime(CLOCK_MONOTONIC, &start);
#endif

NAME::sort(first, last + 1, std::less<\*>(()));

#define MEASURE_TIME
    clock_gettime(CLOCK_MONOTONIC, &dtime);
    dtime.tv_nsec -= start.tv_nsec;
    dtime.tv_sec -= start.tv_sec;
    if (dtime.tv_nsec < 0) {
        dtime.tv_sec --;
        dtime.tv_nsec += 1000000000;
    }
#endif
```
Simulation

Valgrind provides Cachegrind which can simulate a machine running a program and give a number of cache misses and branch mispredictions.

However, it doesn’t consider other activity, including kernel, other processes, TLB misses.
Results: tuned_quicksort is ugly
Results: skewed_introsort is bad
Results: ssssort is good
Results: `std::sort` uses introsort
Conclusion

- We managed to try a lot of tools that we can use in many other areas;
- We found two implementations that give very good results and we proved that with experiments;
- Creating a portable framework for performance measurements would be a very useful project.
Thank you for your attention!

ask questions