Contents

Todo list

Architektur

Speicherzugriffe (memcpy, mram_read ...)

triple buffer

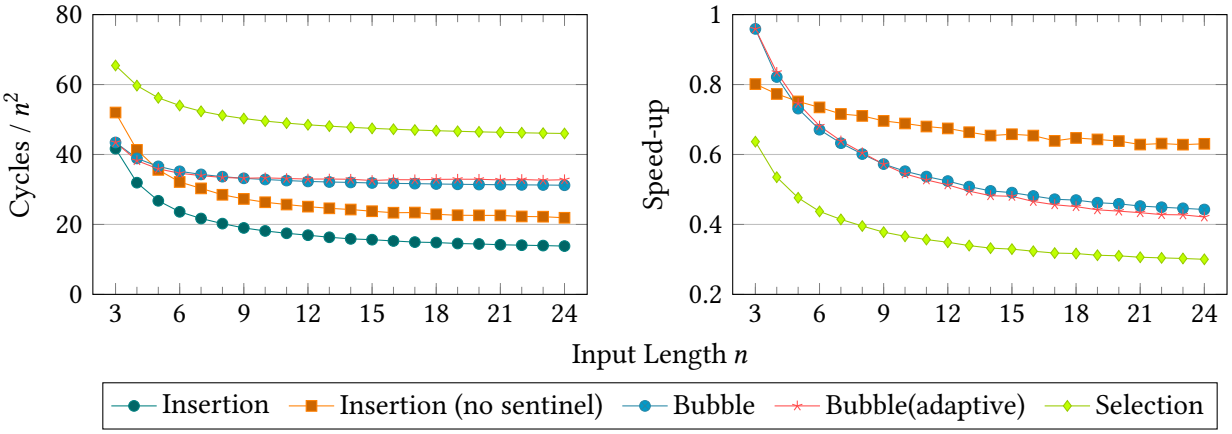

Figure 1: Comparison of sorting algorithms with $O(n^2)$ runtime on a uniform input distribution. The InsertionSorts differ in whether they rely on sentinel values. The adaptive BubbleSort terminates prematurely if no changes were made to the input array during an iteration. The speed-ups are with respect to the InsertionSort relying on sentinel values.

InsertionSort This stable sorting algorithm works by moving the ith element to the left as long as its left neighbour is bigger, assuming that the elements 0 to $i-1$ are already sorted. Even though in both the average case and the worst case, InsertionSort has a runtime of $O(n^2)$, it features quite some advantages: 1. It works in-place, needing only $O(1)$ additional space. 2. It is inherently adaptive: If the input array is mostly or even fully sorted, the runtime drops down to $O(n)$. 3. Its program code is short, lending itself to inlining. 4. The overhead is small. Especially the last two points make InsertionSort a good base algorithm for asymptotically better sorting algorithms to use on very small subarrays.

Beleg?

When moving an element to the left, two checks are needed: Does the left neighbour exist and is it smaller than the element to move? The first check can be omitted through the use of *sentinel values*: If the element at index -1 is at least as small as any value in the input array, the leftwards motion stops there at the latest. Since a DPU has no branch predictor, the slowdown from performing twice as many checks as needed is quite high and lies between 20% and 40% in the relevant input range (Fig. 1).

auf Kom-ex- und implizite Wächter-werte benennen

auf Compilersperenzchen eingehen?

Other known simple sorting algorithm with similar runtime complexity are SelectionSort and BubbleSort. The asymptoticity, however, hides much higher constant factors such that even for as little as three elements InsertionSort is superior (Fig. 1) and should always be used.

ShellSort InsertionSort suffers from small elements at the end of the input array, since those have to be brought to the front through $O(n)$ comparisons and swaps. ShellSort, proposed by Donald L. Shell in 1959 [1], remedies this by doing multiple passes of InsertionSort with different step sizes: In round i with step size h_i, the input array is divided into the subarrays of indices $(0, h_i, 2h_i, \dots)$, $(1, h_i+1, 2h_i+1, \dots)$, $(2, h_i+2, 2h_i+2, \dots)$ and so on which then get sorted individually via InsertionSort. The step size decreases with each round, with the final step size being 1 such that a regular InsertionSort is performed. The individual InsertionSorts should be fast since elements which need to travel long distances already did big jumps.

Finding the right balance between the heightened overhead through multiple InsertionSort passes and the shortened runtime of each InsertionSort pass is subject to research and depends on the cost of the operations (comparing, swapping, looping). Let us first take a quick look on Shell

Belege

Unikosten für DPU analysierend einbauen?

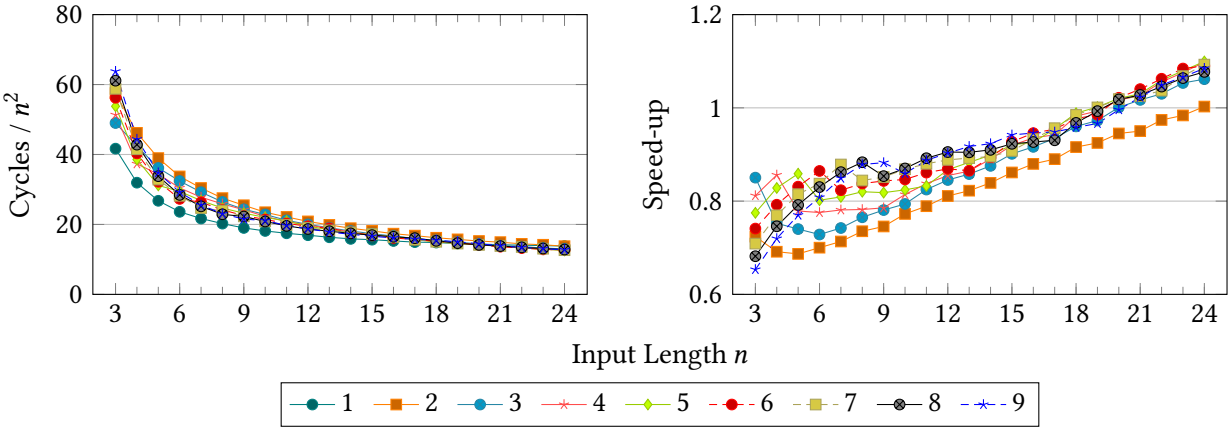

Figure 2: Comparison of InsertionSort (1) and various ShellSorts (2–9) on a uniform input distribution. Each ShellSort does one InsertionSort pass with a step size between 2 and 9 before doing a pass of regular InsertionSort. The speed-ups are with respect to the InsertionSort.

Sort for small input arrays where surely only two rounds with step sizes h_1 and 1 suffice. The previous results on InsertionSort suggest that ShellSort should make use of h_1 sentinel values. Figure 2 shows that the additional rounds starts to pay off at 19 elements for $h_1 \geq 5$, at 20 elements for $h_1 \geq 3$, and at 25 elements for $h_1 = 2$. Bear in mind that these measurements were conducted on a uniform input distribution; if ShellSort is used by another algorithm on a subarray, these thresholds may be even higher or even non-existent due to some degree of presorting.

r0 start

r1 end

r23 return address

```
insertion_sort_nosentinel:
    jleu r0, r1, .LBB2_1  // Continue if array of positive length …
.LBB2_8:
    jump r23  // … else leave the function.
.LBB2_1:
    move r2, r0, true, .LBB2_2  // i ← start; Jump to beginning of outer loop.
.LBB2_5:
    move r4, r5  // ?
.LBB2_7:
    add r2, r2, 4  // i++
    sw r4, 0, r3  // *curr ← to_sort
    jgtu r2, r1, .LBB2_8  // If i > end, terminate.
.LBB2_2:  // Beginning of outer loop.
    lw r3, r2, 0  // to_sort ← *i;
    add r5, r2, -4  // prev ← i - 1
    move r4, r2  // curr ← i
    jltu r5, r0, .LBB2_7  // If prev < start, skip to the next iteration of the outer loop.
    move r5, r2  // (prev + 1) ← i
.LBB2_4:
    lw r6, r5, -4  // *prev
    jleu r6, r3, .LBB2_5  // If *prev > to_sort, terminate inner loop.
    add r4, r5, -4  // Store prev.
    add r7, r5, -8  // Store prev--.
    sw r5, 0, r6  // *curr ← *prev
    move r5, r4  // curr ← prev
    jgeu r7, r0, .LBB2_4  // If prev >= start, continue with the next iteration of the inner l
    jump .LBB2_7  // Continue with the next iteration of the outer loop.

insertion_sort_sentinel:
    jleu r0, r1, .LBB3_1  // Continue if array of positive length …
.LBB3_5:
    jump r23  // … else leave the function.
.LBB3_4:
    add r0, r0, 4  // i++
    sw r3, 0, r2  // *curr ← to_sort
    jgtu r0, r1, .LBB3_5  // If i > end, leave the function.
.LBB3_1:
    lw r2, r0, 0  // to_sort ← *i
    lw r4, r0, -4  // *prev
    move r3, r0  // curr ← i
    jleu r4, r2, .LBB3_4  // If *prev > to_sort, terminate inner loop.
    move r3, r0  // ???
.LBB3_3:
    sw r3, 0, r4  // *curr ← *prev
    lw r4, r3, -8  // *(prev - 1)
    add r3, r3, -4  // curr ← prev
```

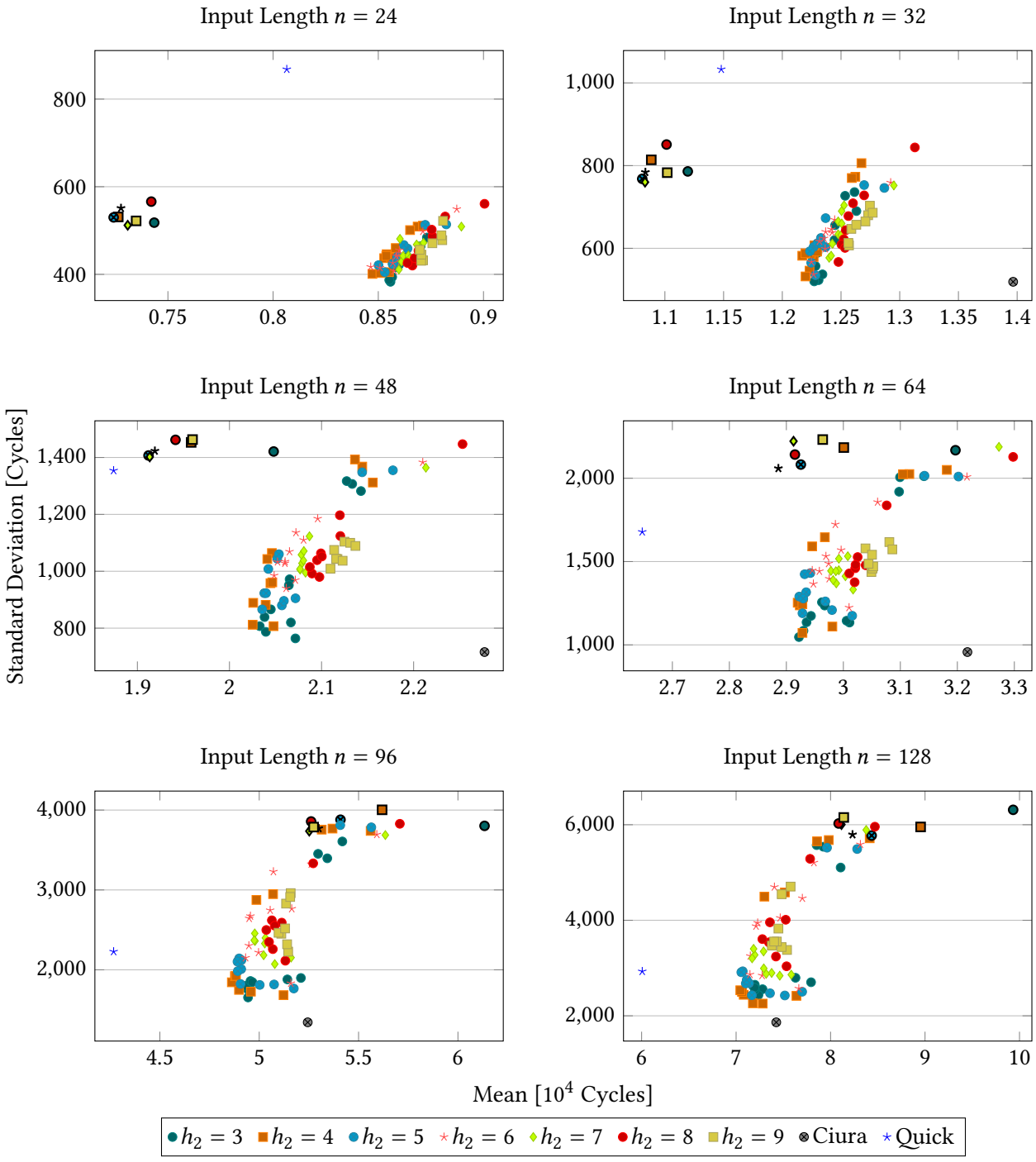
Input Length $n = 24$
Input Length $n = 32$
Input Length $n = 48$
Input Length $n = 64$
Input Length $n = 96$
Input Length $n = 128$
Standard Deviation [Cycles]
Mean [10^4 Cycles]
$h_2 = 3$
$h_2 = 4$
$h_2 = 5$
$h_2 = 6$
$h_2 = 7$
$h_2 = 8$
$h_2 = 9$
Ciura
Quick

Figure 3

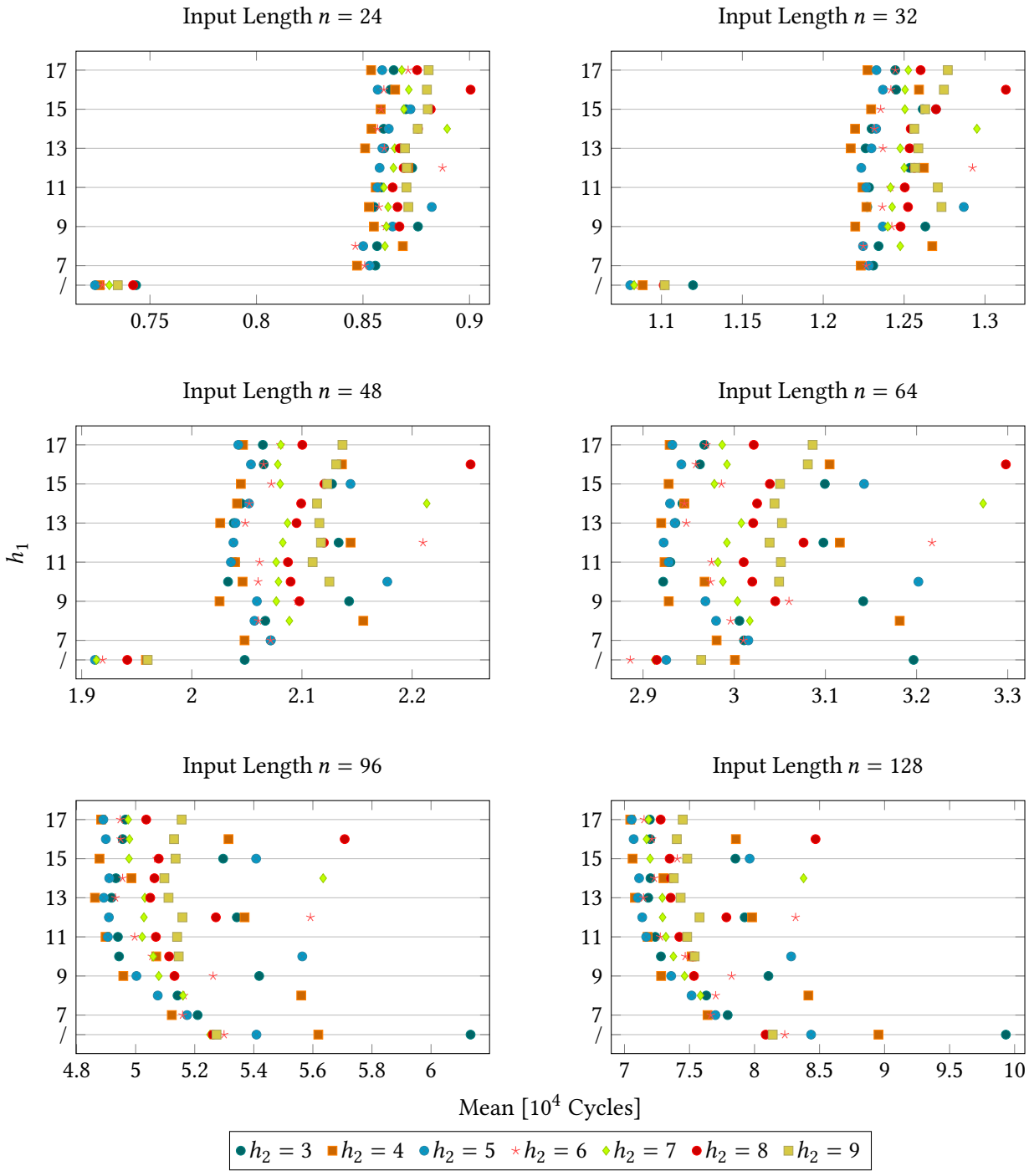
Input Length $n = 24$
Input Length $n = 32$
Input Length $n = 48$
Input Length $n = 64$
Input Length $n = 96$
Input Length $n = 128$
h_1
17
15
13
11
9
7
/
0.75
0.8
0.85
0.9
1.1
1.15
1.2
1.25
1.3
1.9
2
2.1
2.2
2.9
3
3.1
3.2
3.3
4.8
5
5.2
5.4
5.6
5.8
6
7
7.5
8
8.5
9
9.5
10
Mean [10^4 Cycles]
$h_2 = 3$
$h_2 = 4$
$h_2 = 5$
$h_2 = 6$
$h_2 = 7$
$h_2 = 8$
$h_2 = 9$

Figure 4

```
jgtu r4, r2, .LBB3_3  // If *(prev - 1) > to_sort, continue with the next iteration of the
jump .LBB3_4  // Leave inner loop.
```

1 References

[1] Donald L. Shell. 'A high-speed sorting procedure'. In: *Commun. ACM* 2 (1959), pp. 30–32. URL: https://api.semanticscholar.org/CorpusID:28572656.