Tune-up for the Dead-Zone Algorithm

Jorma Tarhio[1] and Bruce W. Watson[2]

[1] Department of Computer Science
Aalto University, Finland
jorma.tarhio@aalto.fi

[2] Department of Information Science
Stellenbosch University, South Africa
bruce@fastar.org

Abstract. We present a number of performance tuning techniques as applied to the Dead-Zone algorithm for exact single (keyword) pattern matching in strings in sequential processing. The tuning techniques presented here are focused on the algorithm skeleton as well as how the shifters are used, and include: removal of some redundant computation, and shifting using 2-grams, among others. Benchmarking results are given for the C implementation in a modern processor without penalties for misaligned memory access.

1 Introduction

String searching is a common task in any software which processes text. The task can be implemented either as an index search, like in web search engines, or as a local search—as in a web browser showing a loaded web page. Here we consider only the latter one where the text to be searched has not been processed beforehand. Formally, the *exact string matching problem* is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in an alphabet Σ, find all the occurrences (including overlapping ones) of P in T. So far, dozens of algorithms have been developed for this problem, see e.g. [5].

We present a number of performance tuning techniques as applied to the Dead-Zone algorithms for exact single pattern matching in strings in sequential processing. The original algorithm is actually a family of algorithms, accommodating numerous possible shifters in a way similar to what the Boyer-Moore family does. Because the Dead-Zone algorithm applies two-way shifting, it is possible to construct inputs for which the algorithm makes fewer comparisons than other comparison-based algorithms.

The tuning techniques presented here are focused on the algorithm skeleton as well as how the shifters are used, and include: removal of some redundant computation (surprisingly, not caught by the optimising compiler), and shifting using 2-grams, among others. Benchmarking results are given for the C implementation. The experiments show that tuning triples the speed of the algorithm.

The rest of the paper is organised as follows. Section 2 present principles of the Dead-Zone algorithms and introduces the base algorithm. Section 3 shows how we optimised the base algorithm. Section 4 gives the results of our practical experiments, and the discussion of Section 5 concludes the article.

2 Background

Here, we only give a brief introduction to the functioning of the Dead-Zone, while some other papers [3,12,16,17] provide a broader picture. Some of the performance details

Jorma Tarhio, Bruce W. Watson: Tune-up for the Dead-Zone Algorithm, pp. 160–167.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

are discussed in [12], which also offers more pointers to various Dead-Zone versions as well as correctness proofs. In particular, that paper gives some simple recursive versions of Dead-Zone before presenting a loop-based (non-recursive) implementation which explicitly maintains a stack for efficiency. That loop-based version, known as *DZ(iter,sh)* in [12], is slightly improved and presented here as Algorithm DZ0—with an explanation below.

Algorithm DZ0 (Dead-Zone)

```
 1  $lo \leftarrow 0; hi \leftarrow n - (m-1)$
 2  $count \leftarrow 0$
 3  push$(0, max)$
 4  while true do
 5     $probe \leftarrow \lfloor (lo + hi)/2 \rfloor$
 6     $i \leftarrow 0$
 7     while $i < m$ and $p_i = t_{probe+i}$ do $i \leftarrow i + 1$
 8     if $i = m$ then $count \leftarrow count + 1$
 9     $kdleft \leftarrow probe - shl[t_{probe}] + 1$
10     $kdright \leftarrow probe + shr[t_{probe+m-1}]$
11     if $lo < kdleft$ then
12        push$(kdright, hi)$
13        $hi \leftarrow kdleft$
14     else
15        $lo \leftarrow kdright$
16        if $lo \geq hi$ then
17           while $top.first \geq top.second$ do pop
18           if $top.second = max$ then return $count$
19           else
20              $lo \leftarrow top.first$
21              $hi \leftarrow top.second$
22              pop
```

As mentioned earlier, rather than recursion, this version of Dead-Zone maintains a stack of 'live-zones'—substrings of the text T still to be considered for matches; each such substring is represented by its beginning index 'lo' (inclusive) and end index 'hi' (not inclusive) [12]. Line 3 pushes a sentinel live-zone onto the stack to make empty-stack detection more efficient, and initialises the lo and hi variables to indicate that the entire string is still a live-zone, keeping in mind that the upper bound is $n - (m - 1)$ because matches cannot occur in the last $m - 1$ symbols.

The outer while loop introduced in line 4 has no guard and is exited when the sentinel element of the stack is popped, indicating there are no more live-zones to consider.

In general, Dead-Zone algorithm variants are divide-and-conquer style—with a resemblance to Quick Sort. Lines 5–8 determines the mid-point/probe of the current live-zone (line 5), then uses a small inner loop (called a match loop) to make a match attempt at that position (lines 6 and 7), and notes any match by incrementing a counter (line 8); as with the SMART framework[1], only the number of matches is tracked as opposed to the positions of all matches.

Line 10 (we return to line 9 shortly) uses a (precomputed) lookup table *shr* giving a *right shift* used to split the current live-zone and compute the *new lo* of the right-hand portion. In this particular version, the lookup table is indexed by the character aligned

[1] https://www.dmi.unict.it/~faro/smart/

with the end of P, namely $t_{probe+m-1}$; this is known as the Horspool shifter [7], and its precomputation is not discussed here. In fact, any Boyer-Moore style shift function could have been used, and this is one of the optimisation opportunities discussed in the next section. Line 9 similarly uses a *symmetrical* left shift table *shl*[2] to give the *new hi* of the left-hand split of the current live-zone. Again, precomputation of that shift table is not discussed here.

Because shifters are symmetrical and *kdleft* is a lower bound pointing to the first dead position, one must be added to it in line 9. Alternatively, this addition could be incorporated into the table *shl*.

Lines 11–13 evaluate whether the newly-determined left-hand portion is empty (test, line 11)—if not, it needs to be explored, and the newly-determined right-hand portion is pushed onto the stack (line 12) for later consideration before proceeding with the left-hand portion (line 13).

Lines 14 onwards are for the case where the left-hand portion *is empty*, meaning that we proceed only with the newly-determined right-hand portion, starting with line 15. Line 16 onwards considers the possibility that this newly-determined right-hand portion is also empty, in which case elements are repeatedly popped from the stack (loop on line 17) until the top-of-stack contains a non-empty live-zone. If the top-of-stack is the sentinel pushed in line 3, the algorithm has fully explored T and it returns (line 18). If not, the top-of-stack contains the live-zone to use and *lo* and *hi* are appropriately updated and that element popped.

Execution then continues (in the live-zone just determined in lines 11–22) at the top of the loop in line 4.

3 Development

Our aim was to develop a faster version of the Dead-Zone algorithm. We tried several local changes to Algorithm DZ0 and evaluated experimentally how they affected the performance. As a result, we ended up suggesting three local optimisations to the Dead-Zone algorithm.

3.1 Elimination of Dead-Zones in the Stack

We noticed that Algorithm DZ0 may sometimes push dead-zones onto the stack. This can be eliminated by adding the test $kdright < hi$ to line 12 before pushing. After this change, the stack contains only live-zones and popping of dead-zones in line 17 can be removed. These changes make the algorithm a bit faster on average. Let us call the modified version Algorithm DZ1. The pseudocode of DZ1 given below shows the changes to Algorithm DZ0.

3.2 Shifting with 2-Grams

Shifting in Algorithms DZ0 and DZ1 is based on single characters according to Horspool's shift [7]. We tried several alternatives for Horspool's shift. It would be possible to apply a different strategy to left shift than to right shift but we decided to use the same strategy to the both directions in order to reduce the number of alternatives.

[2] Usually this is literally a mirror image of the right shift table, Horspool in this case. That is not a requirement and other left shifters may be used.

```
Algorithm DZ1 (Changes to DZ0)
11  if $lo < kdleft$ then
12      if $kdright < hi$ then push($kdright, hi$)
13      $hi \leftarrow kdleft$
14  else
15      $lo \leftarrow kdright$
16      if $lo \geq hi$ then
17
18          if $top.second = max$ then return $count$
```

Sunday's shift [14] is a natural choice for the Dead-Zone algorithm, because it is applied after exiting the match loop testing an alignment window $t_{probe} \cdots t_{probe+m-1}$ in the text against the pattern. The test characters of shift $t_{probe-1}$ and $t_{probe+m}$ are outside the alignment window, whereas the test character is the first/last character of the alignment window in Horspool's shift. Thus the maximal shift of Sunday is $m+1$ to the both directions, i.e. one more than Horspool's maximal shift. On average, DZ1 with Sunday's shift runs slightly faster than DZ1.

Shifting based on 2-grams is a better choice for making the algorithm more efficient. The original lookup tables *shl* and *shr* of DZ1 are replaced by new two-dimensional lookup tables *shl2* and *shr2*, respectively. We tried three 2-gram shifters, in which the locations of the test 2-grams are different as well as the preprocessing of the lookup tables.

First we tried the Zhu–Takaoka shift [18]. The original method consists of two shift functions like the Boyer–Moore algorithm [2]. We applied only the 2-gram shift based on the occurrence heuristic (a.k.a. the bad character heuristic). We denote this shifter by ZT. The test 2-grams are the first and last 2-gram of the alignment window. The maximal shift of ZT is m.

Next we tested the Berry–Ravindran (BR) shift[3] [1] which is an extension of Sunday's shift. The tested 2-grams $t_{j-2}t_{j-1}$ and $t_{j+m}t_{j+m+1}$ ($j = probe$) are outside the alignment window. The maximal shift of BR is $m+2$.

The third 2-gram shifter is BRX introduced by Kalsi et al. [9]. BRX is an intermediate approach of ZT and BR. The test 2-grams are $t_{j-1}t_j$ and $t_{j+m-1}t_{j+m}$, $j = probe$. The maximal shift of BRX is $m+1$. Figure 1 shows the locations of the test 2-grams of ZT, BRX, and BR in an alignment of a pattern.

In our experiments (see Section 4), the three 2-gram shifters gave a significant speed-up over Algorithm DZ1. BR was the slowest. ZT and BRX were almost equally good, but BRX was the winner in case of short DNA patterns. We selected BRX for further development.

[3] Mauch [11] (and related publications) was the first to apply the BR shift to the Dead-Zone algorithm.

```
  P:     acctcg
  T: acccgtatgactta
 ZT:     xx  xx
BRX:    xx    xx
 BR:   xx      xx
```

Figure 1. Locations of test 2-grams of ZT, BRX, and BR in an alignment.

Let us consider detailed conditions for the shift tables of BRX. The test 2-qram $y = y_1 y_2$ of the right-hand shift is $t_{probe+m-1} t_{probe+m}$. If y is present in P, then

$$shr2[y] = m - \max(i \mid y = p_i p_{i+1}) - 1$$

Otherwise $shr2[y]$ is $m + 1$ if $y_2 \neq p_0$ and m if $y_2 = p_0$. The left-hand shift is symmetrical to the right-hand shift. The test 2-qram $x = x_1 x_2$ is $t_{probe-1} t_{probe}$. If x is present in P, then

$$shl2[x] = \min(i \mid x = p_i p_{i+1}) + 1$$

Otherwise $shl2[x]$ is $m+1$ if $x_1 \neq p_{m-1}$ and m if $x_1 = p_{m-1}$. Based on these conditions, it is straightforward to program the preprocessing of *shr2* and *shl2*.

We modified the BRX shifter further. Instead of two-dimensional shift tables, we implemented the handling of 2-grams as 16-bit entities. This version is called Algorithm DZ2. Notation $q(x, h)$ refers to a h-gram starting at x. At the implementation level $q(x, 2)$ is `*((uint16_t*)x)`. The pseudocode of DZ2 given below shows the changes to Algorithm DZ1.

Algorithm DZ2 (Changes to DZ1)

```
9    $kdleft \leftarrow probe - shl2[q(t_{probe-1}, 2)] + 1$
10   $kdright \leftarrow probe + shr2[q(t_{probe+m-1}, 2)]$
```

3.3 Guard Test

Guard test [7,13] is a widely used technique to speed-up string matching. The idea is to test certain pattern positions before entering a match loop. Guard test is a representative of a general optimisation technique called loop peeling, where a number of iterations are moved in front of the loop. As a result, the computation becomes faster because of fewer loop tests.

We decided to try such a guard test where the first q-gram *try* of an alignment in the text is compared with *prefix*, the first q-gram of the pattern. As a result, the match loop is entered more seldom. We tried values $q = 2$ and 4. We decided to apply the latter, because it performed better. Let us call the modified version Algorithm DZ3. The pseudocode of DZ3 given below shows the changes to Algorithm DZ2.

Algorithm DZ3 (Changes to DZ2)

```
5b   $try \leftarrow q(t_{probe}, 4)$
5c    if $try = prefix$ then
6        $i \leftarrow 4$
7        while $i < m$ and $p_i = t_{probe+i}$ do $i \leftarrow i + 1$
8        if $i = m$ then $count \leftarrow count + 1$
```

Algorithm DZ3 in the present form does not work for patterns shorter than four characters. However, it is trivial to add separate code for them if necessary.

Beyond the Dead-Zone algorithm, the guard test with a q-gram might improve the performance of some other algorithms as well. Testing of q-grams as entities has been earlier used by Faro and Külekci [4] and Khan [10]. If wider q-grams than four characters are applied to long patterns, separate code is necessary for short patterns.

With old processors, there is a performance penalty for reading q-grams at misaligned memory locations, i.e. the q-gram does not start at a word boundary. This penalty decrease the benefit of processing q-grams as entities for shifting or guarding. However, for newer processor microarchitectures of Intel starting from Sandy Bridge and Nehalem, there is no such penalty [6].

4 Experiments

The experiments were run on Intel Core i7-4578U with 4 MB L3 cache and 16 GB RAM; this CPU has a Haswell microarchitecture which is subsequent to Sandy Bridge and therefore has none of the misaligned access performance penalties mentioned above. Algorithms were written in the C programming language and compiled with gcc 5.4.0 using the O3 optimisation level. Testing was done in the framework of Hume and Sunday [8]. We used two texts: English (four concatenated copies of the KJV Bible, totaling 16.2 MB) and DNA (four concatenated copies of the genome of E. Coli, totaling 18.6 MB) for testing. The base texts were taken from the SMART corpus. Because of the irregular scanning order, the Dead-Zone algorithms benefit from cache more than other algorithms. This was noticeable for texts shorter than 6 MB in our test setting, and therefore we decided to use longer texts. Sets of patterns of lengths 5, 10, and 20 were randomly taken from the both texts. Each set contains 200 patterns. The running times of 200 patterns in Table 1 are averages of 100 runs excluding the preprocessing time. For the 2-gram shifters, preprocessing took about 10 ms per 200 patterns. We used Horspool's algorithm (Hor) [7] and Sbndm4b [15] as reference methods. Hor is a representative of classical algorithms and Sbndm4b is an example of a fairly efficient algorithm. The code of Hor was taken from the SMART repository.

Table 1. Running times (in seconds) of algorithms.

	English, m			DNA, m		
Alg.	5	10	20	5	10	20
DZ0	5.25	3.64	2.77	13.43	10.74	10.27
DZ1	4.99	3.56	2.73	12.76	10.26	9.83
DZ1s	4.39	3.20	2.47	12.38	10.99	10.55
DZ1br	3.67	2.19	1.43	8.96	6.62	5.30
DZ1zt	3.19	1.97	1.33	9.24	5.26	3.63
DZ1brx	2.98	1.87	1.26	6.97	4.81	3.57
DZ2	2.71	1.72	1.16	6.47	4.50	3.38
DZ3	2.12	1.41	0.99	4.09	3.02	2.37
Hor	4.17	2.41	1.49	10.30	7.39	7.10
Sbndm4b	1.18	0.42	0.30	1.50	0.63	0.45

For Algorithm DZ1 using Horspool's shift we tried four alternative shifters:

1. DZ1s: Sunday [14]
2. DZ1br: Berry–Ravindran [1]
3. DZ1zt: Zhu–Takaoka [18] (occurrence shift)
4. DZ1brx: Kalsi et al. [9]

Algorithm DZ1s ran slightly faster than DZ1 for English data. Although the average shift of DZ1s is longer than that of DZ1, DZ1s was slower than DZ1 for DNA patterns of 10 and 20 characters. All the 2-gram approaches DZ1br, DZ1zt, and DZ1brx were significantly faster than the single character shifters DZ1 and DZ1s.

Algorithm DZ1br was the slowest of the 2-gram shifters. This is obvious for DNA because the average shift of DZ1br is shorter than that of DZ1zt for $m > 7$. The reason for the poor performance of DZ1br for English data is partly due to appearances of common characters like space as p_0 or p_{m-1} which decreases the average length of shift (see justification in [9]). DZ1brx was slightly better than DZ1zt for English data, but the former was a clear winner in the case of short DNA patterns.

Algorithm DZ3 was clearly faster than Hor but left noticeably behind Sbndm4b. Comparison of Dead-Zone algorithms and efficient left-to-right algorithms is somewhat unfair in sequential processing, because the former algorithms contain more bookkeeping and they do not benefit from locality as much as the latter ones.

Table 2. Speed-ups of algorithms (Alg. DZ0 is one).

	English, m			DNA, m			
Alg.	5	10	20	5	10	20	Avg.
DZ0	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DZ1	1.05	1.02	1.01	1.05	1.05	1.04	1.04
DZ1s	1.20	1.14	1.12	1.08	0.98	0.97	1.08
DZ1br	1.43	1.66	1.93	1.50	1.62	1.94	1.68
DZ1zt	1.64	1.85	2.08	1.45	2.04	2.83	1.98
DZ1brx	1.76	1.95	2.20	1.93	2.23	2.88	2.16
DZ2	1.94	2.11	2.38	2.08	2.38	3.03	2.32
DZ3	2.48	2.59	2.79	3.28	3.56	4.34	3.17

Table 2 shows speed-ups of the developed Dead-Zone versions against Algorithm DZ0. Algorithm DZ3 clearly tripled the speed of the original algorithm DZ0 on average. The gain was larger for DNA than for English.

5 Concluding Remarks

Although the Dead-Zone algorithm is clearly oriented to parallel processing, we managed to substantially improve its performance in sequential processing. Only some of the optimisations are unique to the Dead-Zone algorithm, and the others could be used to benefit many of the other well-known algorithms when they are not already used. In particular, multi-gram shifting is an interesting tradeoff of memory (for shift tables) against performance, while the guard test optimisation could be applied in most Boyer-Moore style algorithms. It is somewhat surprising that the guard test optimisation is not automatically part of gcc's O3 level, as such optimisations are well known in the compiler literature. In short, despite continuous compiler and algorithmic improvement, there remain interesting opportunities for skilled programmers to manually tune implementations.

Some optimisations may be sensitive to misaligned memory accesses—though this was not the case on the benchmarking system used for this paper. This indicates that future work could include optimisations on other architectures where misalignment gives a performance penalty, or on alternative architectures such as the Arm. Additional future work includes using this paper's optimisations on multiple-keyword Dead-Zone.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop 1999, Prague, Czech Republic, July 8-9, 1999, 1999, pp. 16–28.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
3. J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, É. Prieur-Gaston, and B. W. Watson: *Three strategies for the dead-zone string matching algorithm*, in Prague Stringology Conference 2018, Prague, Czech Republic, August 27-28, 2018, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2018, pp. 117–128.
4. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013, 2013, pp. 113–121.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
6. A. Fog: *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, tech. rep., Technical University of Denmark, www.agner.org/optimize/microarchitecture.pdf, 2020.
7. R. N. Horspool: *Practical fast searching in strings.* Softw., Pract. Exper., 10(6) 1980, pp. 501–506.
8. A. Hume and D. Sunday: *Fast string searching.* Softw., Pract. Exper., 21(11) 1991, pp. 1221–1248.
9. P. Kalsi, H. Peltola, and J. Tarhio: *Comparison of exact string matching algorithms for biological sequences*, in Bioinformatics Research and Development, Second International Conference, BIRD 2008, Vienna, Austria, July 7-9, 2008, Proceedings, 2008, pp. 417–426.
10. M. A. Khan: *A transformation for optimizing string-matching algorithms for long patterns.* Comput. J., 59(12) 2016, pp. 1749–1759.
11. M. Mauch: *An Investigation of Dead-Zone Pattern Matching Algorithms*, Master's thesis, Stellenbosch University, South Africa, 2016.
12. M. Mauch, D. G. Kourie, B. W. Watson, and T. Strauss: *Performance assessment of dead-zone single keyword pattern matching*, in 2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012, J. H. Kroeze and R. de Villiers, eds., ACM, 2012, pp. 59–68.
13. T. Raita: *On guards and symbol dependencies in substring search.* Softw., Pract. Exper., 29(11) 1999, pp. 931–941.
14. D. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
15. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
16. B. W. Watson, L. G. Cleophas, and D. G. Kourie: *Using correctness-by-construction to derive dead-zone algorithms*, in Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 84–95.
17. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.
18. R. F. Zhu and T. Takaoka: *A technique for two-dimensional pattern matching.* Commun. ACM, 32(9) 1989, pp. 1110–1120.