Reordering Finite Automata States for Fast String Recognition

E. Ketcha Ngassam[a], Derrick G. Kourie[b], and Bruce W. Watson[b]

[a] School of Computing, University of South Africa,
Pretoria 0003, South Africa
[b] Department of Computer Science, University of Pretoria,
Pretoria 0002, South Africa

e-mail: [a]ngassek@unisa.ac.za, [b]{dkourie, bwatson}@cs.up.ac.za

Abstract. The spatial and temporal locality of reference on which cache memory relies to minimize cache swaps, is exploited to design a new algorithm for finite automaton string recognition. It is shown that the algorithm, referred to as the state reordering algorithm, outperforms the traditional table-driven algorithm for strings that tend to repeatedly access the same set of states.

Keywords: Automata, Implementation, Performance, State Reordering, Cache Locality of Reference

1 Introduction

Traditionally, finite automata (FAs) are implemented using the table-driven (TD) algorithm extensively discussed in [1]. In this case, the processing time of the recognizer is memory load dependant in the sense that for automata of considerable size[1], the time taken to process a string not only depends on the length of the string but also on the time taken to do a lookup in the transition matrix.

In [2, 3], we reported on investigations based on hardcoded FAs which appeared to be faster than the TD algorithm, but only for automata of relatively small sizes[2]. Further investigations revealed that, although memory load and string length are major processing time factors, the kind of string being tested for acceptance is also critical. In effect, no matter the size of the string being tested for acceptance and the size of the automaton upon which the recognizer relies, a string that drives the automaton into a set of 'sink' states throughout the recognition process is always processed at optimum due to computer's cache memory [4]. In such kinds of strings, the hardcoded algorithm outperforms its TD counterpart. This is explained by the fact that the instructions that makeup the hardcoded algorithm always remain in cache—hence the fast processing speed. In this regard, cache memory plays an important role in determining the efficiency of FA-based string recognition algorithms.

[1] In this paper, we use automaton size to mean the number of states of the automaton. The two terms are used interchangeably.

[2] In fact experiments revealed that hardcoded algorithm is faster than TD for FAs of size up to about 360 states on an Intel Pentium 4. This was true for alphabet sizes of up to about 50 symbols.

Cache memory operation is based on what is sometimes referred to as the principle of *temporal and spatial locality of reference*. Since data/instructions are fetched from memory in blocks, the temporal locality of reference refers to the premise that there is a strong chance that the same data/instruction will be used in the near future. Similarly, spatial locality of reference refers to the premise that there is a strong chance that other data within a given block will be fetched in the immediate future. These two principles are of importance in the design and implementation of efficient algorithms. Moreover, the nature of the cache itself guarantees that data found in cache is processed faster than data residing in the main memory. Page swaps into cache occur when data being sought is not in cache and the cache is full. In this case, a policy such as that of LRU (least recently used) data is normally used to determine what is to be swapped out. For more information on cache, refer to [5].

In this paper, we provide an alternative algorithm for string recognition, referred to as the State Reordering (SR) algorithm that makes use of the spatial and temporal locality principles. The algorithm reorders the states of the original automaton according to the string being processed. Only states needed are reorganized in memory.

In certain circumstances, the reordering increases the probability of reusing the chunks of data already present in the cache. The evidence suggests that our algorithm will outperform the TD algorithm for large automata when processing long sequences that exercise a limited number of states. The provision is that the strings are long enough to amortize the cost of reordering the states. This would be the case, for example, in a network intrusion detection system, where a continuous stream of data is being processed by an FA-based system.

The structure of the rest of this paper is as follows. In Section 2 we present and explain the SR algorithm. Section 3 assesses it from a theoretical perspective. Section 4 deals with the experimental comparison of SR and TD. Finally, in section 5, the conclusion and further directions for this work are offered.

2 The State Reordering Algorithm

In this section, we present the new SR algorithm and provide a theoretical analysis of the algorithm based on strings of considerable length. The conditions under which the algorithm appears to be most efficient are also discussed. We further the analysis by providing a class of strings that can benefit from our algorithm provided that the set of states visited remains unchanged.

proc $tdRecognizer(table, inString)$
$; state, index := 0, 0$
do $(index < inString.length()) \wedge (state \geq 0) \rightarrow$
    $state, index := table[state][inString[index]], index + 1$
od

Figure 1: Table-driven string recognizer

It is clear from the pseudocode of the TD algorithm (see Figure 1) that its access

to the transition table in memory is entirely dependent on the string being examined. Since data is fetched by the processor from memory in chunks, the arbitrary organization of the table's entries results in frequent cache misses in the next cycle. Put differently, the probability of finding the desired datum from the cache is relatively low. The processor is then forced to perform a page swap in order to get the desired entry. This approach may result in inefficiencies when the table is considerably large.

Figure 2 provides a high-level specification of the SR algorithm. Just as in figure 1, the transition table and the input string ($inString$) are provided as parameters. Also provided as parameters are: the start address ($start$) where information about reordered states is stored; and an indication of the amount of space to be reserved for each reordered state ($size$). By a reordered state, we mean a state (as represented by a row in the original transition table) whose information have been copied (and modified — see below) into a specially reserved place in memory, indicated by the dynamic two-dimensional array, $srTable$. The main loop consists of an alternation- (i.e. if-) statement, and an assignment statement to increment the value of the current index into $inString$. The loop condition corresponds identically to that of the TD algorithm. The alternation statement has two guards: the first deals with a transition to the next state when the current state has not yet been reordered; and the second deals with a transition from a reordered state.

The algorithm uses an auxiliary array, $srMap$. The invariant of the algorithm's main loop:

$\forall i : [0, n) \cdot srMap[i] = k \wedge k \geq 0 \Leftrightarrow k \in [0, pos) \wedge isReordered(i, k, index - 1)$

articulates the nature of $srMap$, namely that the i^{th} entry of $srMap$ is a positive value, k, if and only if k indexes an entry in $srTable$ (i.e. $k \in [0, pos)$) and that "the k^{th} entry is a reordered state that corresponds to the i^{th} row in the original transition table". The predicate $isReordered(i, k, index - 1)$ is an assertion that corresponds to the words in quotes in the previous sentence, as will be discussed below.

The variable pos holds the index of the next $srTable$ entry to be created in memory. Thus, the first statement of the first guarded command assigns pos to $srMap[state]$, where $state$ is the current state, and the next symbol to be accessed is $inString[index]$. The variable $nextB$ points to the next memory address where space for the entry $srTable[pos]$ is to be allocated. The second statement of the first guarded command allocates the required memory for the $srTable[pos]$ entry, and the third statement copies the transition table values for row $state$ over into a row at $srTable[pos]$.

However, each entry $srTable[k][j]$ is required to have the following property: if its value, say m, is less than the total number of states, n, this should be construed to mean that if symbol j is encountered when in reordered state k, then a transition is to be made to state m where m is *not* a reordered state. However, if m is indeed greater or equal to the total number of states, then this should be construed to mean that the transition in reordered state k upon encountering symbol j is to the reordered state in $srTable[m - n]$. Thus, each time a reordered state is added into memory, it is necessary to check all reordered state entries to re-establish this property. An inner double loop in the first guarded command achieves this objective. Note that the predicate $isReordered(i, k, index - 1)$ is consistent with this property required of $srTable$ entries. It relies on the existence of a set $visited(p)$ which designates the set of all states visited when recognizing the first p elements of the string $inString$.

```
{Assume n is the number of states and a is the alphabet size}
proc srRecognizer(table, inString, start, size)
; srMap[0..n − 1] : = −1
; nextB, state, index, pos : = start, 0, 0, 0
{ Invariant ≜ (∀i : [0, n) · srMap[i] = k ∧ k ≥ 0 ⇔
              k ∈ [0, pos) ∧ isReordered(i, k, index − 1)
  isReordered(i, k, p) ≜ ∀j : [0, a) · m = srTable[k][j] ⇒
              ((m < n ∧ table[i][j] = m ⇔ m ∉ visited(p))
               ∨ (m ≥ n ⇔ ∃r : [0, n) · srMap[r] = m − n ∧ r ∈ visited(p)))
}
;do (index < inString.length() ∧ state ≥ 0) →
    if state < n →
       srMap[state] : = pos
       ; srTable[pos] : = malloc(nextB, size)
       ; srTable[pos][0..a − 1] : = table[state][0..a − 1]
       ; k, j : = 0, 0
       do k ≤ pos →
          do j < a →
             m : = srTable[k][j]
             ; if m < n ∧ srMap[m] < 0 → skip{m ∉ visited(index)}
             [] m < n ∧ srMap[m] ≥ 0 → srTable[k][j] : = srMap[m] + n
             [] m ≥ n → skip{m already updated}
             fi
             ; j : = j + 1
          od
          ; k : = k + 1
       od
       ; state, pos, nextB : = srTable[pos][inString[index]], pos + 1, nextB + size
    [] state ≥ n → state : = srTable[state − n][inString[index]]
    fi
    ; index : = index + 1
od
```

Figure 2: The state reordering string recognizer

The double loop is followed by assignments to update *state*, *pos* and *nextB*. If this new value of *state* turns out to be in the interval $[0, n)$ then it represents a transition to a non-reordered state, and will be dealt with in the next loop iteration by the first guard, in the way just described. However, if *state* turns out to be $\geq n$ then it will be dealt with in the next iteration by the second guard.

This second guard uses *srTable* to perform the recognition of the string being tested for acceptance, as if it were the transition table in the conventional TD algorithm, but correcting, of course, for the offset by n in the *state*'s value. In fact, at every iteration of the loop, whenever the next state is a reordered one, then this

second guard's statement is executed, followed by the final statement to increment the *index* value.

The SR algorithm is thus subdivided into two parts: the *reordering* section, represented by the first guard's body, in which a state that has not been created is reordered—i.e. inserted into the $srTable$; and what we shall call the *hot-spot* section, represented by the second guard.

At first sight, the SR algorithm might appear to be less efficient than the TD version, due to the various tests that have to be made at each iteration as well as to the work done to create new "reordered" states. The SR algorithm would obviously be at a disadvantage in cases where, for a relatively large number of loop iterations, the *reordering* path is followed, since the time taken to allocate and copy memory will hamper the overall processing time. However, as a result of reordering, previously used states are organized contiguously in memory in the same order in which they are first traversed. This could be advantageous if it reduced the number of cache misses in iterations where the *hot-spot* is executed. We defer further discussion about these matters to section 3

A practical example of the SR algorithm is shown in the subsection below.

2.1 An illustrative example

Consider an automaton $M(s_0, \Sigma, Q, F, \delta)$ where $s_0 = 0$, $\Sigma = \{a, b, c\}$,
$Q = F = \{0, 1, 2, 3, 4, 5, 6\}$, and δ defined by a two-dimensional array, given below. This automaton is *partially* represented in Figure 3, in that it only shows transitions that will be followed when the string *abcbaabcbaabcba* is being recognized. The strings

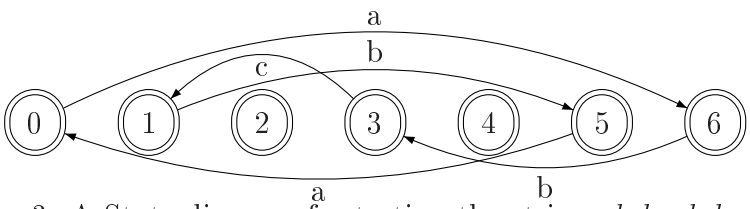

Figure 3: A State diagram for testing the string *abcbaabcbaabcba*

abcbaabcbaabcba can be processed using the SR algorithm as follows:

Initial phase:

After initialization the following holds:
$table = \{\{6, 3, 1\}, \{2, 5, 4\}, \{1, 1, 2\}, \{3, 2, 1\}, \{4, 6, 0\}, \{0, 1, 3\}, \{1, 3, 5\}\}$
(Thus, $\delta(0, a) = 6$, $\delta(0, b) = 3$, etc.)
$inString = abcbaabcbaabcba$
$inString.length() = 15$
$srMap = \{-1, -1, -1, -1, -1, -1, -1\}$.
$nextB, state, index, pos := start, 0, 0, 0$

The first iteration:

At this stage, all the conditions to enter the loop are satisfied. Therefore, the loop is executed. A test is made on $srMap[state]$ to see whether the state has been created or not. For $state = 0$, the first guard is selected and a new state has to be created in memory. This results in the following:
$srMap[0] = 0$, that is the old state 0 will occupy the first position in the new memory space.
The variable *size* represents the memory required to store a state. It depends on the alphabet size (3 for the present example).
The instructions: $srTable[pos] := malloc(nextB, size)$ and $srTable[0][0..2] := table[state][0..2]$ are then executed to produce $srTable = \{\{6, 3, 1\}\}$
The double loop tests whether any entry in the $srTable$ has been reordered to date. None has, so $srTable$ is left unaltered.
The new value of *state* is 6, *pos* becomes 1 and *index* becomes 1

Later iterations

Suppose the substring *abcba* has already been processed. If the double inner loop was not part of the algorithm then $srTable$ would simply be the following:
$\{\{6, 3, 1\}, \{1, 3, 5\}, \{3, 2, 1\}, \{2, 5, 4\}, \{0, 1, 3\}\}$. However, the double inner loop has to make sure that the entries in the new location are distinguished from those of the old location. Therefore, to avoid conflict of states, the double inner loop adds $n = 7$ to all reordered states.
This results in: $srTable = \{\{8, 9, 10\}, \{10, 9, 11\}, \{9, 2, 10\}, \{2, 11, 4\}, \{7, 10, 9\}\}$.
Therefore, for the processing of the string up to this point, only four of the six automaton states were visited. It can easily be seen that the remaining part of the string, that is *abcbaabcba*, involves the traversal of these reordered states only. Thus the remaining string is processed at *hot-spot*.

Before testing the SR algorithm empirically, it is of interest to assess theoretically how the SR and TD algorithms are likely to perform relative to one another. Such task is undertaken in the following subsection.

3 A theoretical assessment

In cross-comparing these algorithms, we rely on the fact that data in cache is processed faster than the data that is in main memory. Furthermore, when data is organized in a contiguous fashion and data items are accessed sequentially, the number of page swaps is minimized. By contrast, when data is accessed in a disorganized or random fashion, the number of cache swaps is high.

Now, as a matter of fact, ultimately neither the TD nor the SR algorithms can of themselves directly influence the way in which cache is used. They are "victims", as it were, of the strings that they are required to recognize. The following is a broad classification of the kinds of scenarios that could arise.

1. If an input string continuously drives an algorithm through a relatively small number of states such that these all remain permanently in cache, then both algorithms function optimally. Even if the input string is relatively long, the

time taken to process a single symbol is optimized. Of course, in such a case, the SR algorithm is a poor one, since it needlessly incurs the initial setup cost during the reordering phase.

2. If the input string drives an algorithm through a somewhat larger number of states, such that cache swaps have to be made, then the question is whether these cache swaps are at a minimum. Again, this behaviour is entirely dependent on the input string.

   (a) Pathological strings could be constructed to induce worst case behaviour for both the TD and SR algorithms, where as many string symbols as possible induce a transition to a state that is not currently in cache.

   (b) Likewise, well behaved string examples could be constructed where state transitions are nicely ordered to progress from row to row in the original transition table.

   In both these extreme situations, TD would perform better than SR, since SR would again incur, without any real gain, the state reordering setup cost.

3. Under the previous scenario (i.e. where a large number of states are traversed), the SR algorithm could potentially acquire an advantage over the TD algorithm if the input string exhibited the following characteristics:

   (a) the string tended to repeatedly exercise the same subset of states; where

   (b) these states were fairly widely distributed over the transition table rows, thus causing many cache misses under TD; but

   (c) where the states were contiguously placed in $srTable$ because the order of their initial usage reflected their later usage and order.

   It is easy to see that under these circumstances, the hot-spot of the SR algorithm would repeatedly be exercised in a way that minimized cache swaps, while the TD algorithm would incur a high number of cache swaps.

The claim made in 3 is rather general. It does not attempt to quantify how many reorderings should take place, how many times the hot-spot should be exercised, how long the input string should be, how rows in the transition table should be ordered, etc. Clearly all of these factors could influence the extent to which SR improves over TD. Indeed, at this point, it is not even clear whether, under practical conditions, the cost of state reordering is ever really likely to pay off. In the next section, experiments are described that offer some insights into these matters.

4 Experiments and Results

Various experiments were conducted on a 512MB Intel Pentium 4 machine, having two levels of cache memory (L1 and L2). The L1 data cache has a capacity of 8KB, with a speed of 2ccs. The L2 cache is bigger and can hold up to 256KB of data and instructions with a relative speed of approximately 6ccs. Data is fetched from memory in chunks of 64 bytes. During initial program execution, if reference is made

to a data item outside of cache, another chunk is fetched until both L1 and L2 cache are full. A subsequent fetch of data not residing in either cache, results in a page swap of memory data with data in the lowest cache. The data to be swapped out from cache is determined by the "Least Recently Used Data" policy.

The SR algorithm was implemented in the NASM assembly language under the Linux operating system. The TD algorithm was originally implemented in C++, with the optimizer (O3) turned on. Its NASM implementation was also provided after several early experiments. The intention was to ensure that the SR algorithm did not enjoy some hidden advantage because of being implemented in an assembler language. It turned out that the NASM version of TD was indeed slightly faster than its C++ implementation for automata larger than about 3000 states. However, the difference was so small that the overall results of our findings apply, no matter which TD implementation is considered.

For the present experiment, 100 automata of size $n = 125, 250, 325, \cdots, 12500$ were generated, based on 10 alphabet symbols. The transition table of each automaton was randomly constructed in the following sense:

- Firstly, for each row, $i : [0, n-2]$, a column $j : [0, a)$ (corresponding to some alphabet symbol) is randomly selected. This column is assigned the next state transition value $i + 1$. This ensures that there is at least one string of length $n - 1$ that will traverse every state of the FA. We shall refer to this string as the *root* string of the particular automaton.

- Next, all remaining cells of the table are assigned a random value in the range $[0, n-1]$.

Considered graphically, this means that each node in the FA graph has a transition to the next state on some random symbol, as well as a transition on each of the remaining 9 alphabet symbols to some random state.

For each automaton, a random string of size $n - 1$ was generated. This was replicated 4 times to produce an input string of length $4n - 4$ consisting of 4 identical segments. Each of the algorithms was required to use the randomly generated automaton of size n to recognize such a string, resulting in 100 runs of each algorithm.

Before discussing the timing results, consider the information presented in table 1. The table gives an overview of the rate at which state reordering was found to occur when the SR algorithm was run. Data is given as a percentage of the total number of states in each particular run. The first column relates to the full string that was processed, the second column, to the first segment, etc. Thus, after processing the first segment, on average a little more than 60% of the states are reordered. Note that these observations lie in a fairly narrow band, between about 57% and 63%. As a matter of fact, when the number of reordered states is plotted against the automaton size (not provided here), a very distinct linear trend is observed. However, in the case of segments 2 to 4, no obvious trend is observed in relation to automaton size. Nevertheless, the average number of reordered states declines steadily from about 16% in the case of segment 2, to almost 0% in the case of segment 4. Overall, about 80% of states were reordered, on average.

These results are broadly in line with expectation. In processing the first $n - 1$ symbols, roughly 60% of the states are reordered, meaning that they are located

	Full String	Segment 1	Segment 2	Segment 3	Segment 4
Maximum	95.20%	62.67%	24.80%	9.17%	2.78%
Minimum	62.42%	56.80%	0.40%	0.00%	0.00%
Average	79.06%	61.29%	16.11%	1.60%	0.06%

Table 1: Rate of Reordered State Generation

contiguously in memory in order of first usage. In this sense, the data is optimized in terms of the spatial locality of reference principle. Later segments trigger progressively fewer state reorderings, and consequently spend more time in the hot-spot part of the code. If these later segments were to traverse the reordered states in *exactly* the same sequence as the first segment, then the probability would be relatively high of accessing spatially localized data, and hence of triggering few cache swaps. Of course, this will only happen in the unlikely event that segment 2 (and therefore also 3 and 4) happen to start off in state 0.

The experiment above has not been designed to specifically generate this "best case" scenario. Rather, it is far more likely that these later segments will start off in some other random state. Nevertheless, on the evidence of table 1, an increasingly large proportion of segment 2 to 4 processing is via the hot-spot. In fact, even in the case of the first segment's processing, about 40% of the iterations were through the hot-spot. Whether this translates into time gains as a result of frequently accessing spatially localized data (and therefore having fewer cache swaps), cannot be predicted *a priori*. To this end, we require the timing data derived from running the respective algorithms.

For the purposes of recording timing data, each algorithm was invoked 50 times for each set of input, and the processing time was recorded in clock cycles (ccs). For further analysis, we relied on the minimum of these 50 observations. (This was because the experience of earlier studies, which had shown that occasionally, outlier data is generated that distorts the average and that is apparently attributable to OS and CPU overheads.)

The results showed that state reordering is too expensive to provide such a short-term payoff. In fact, the cost is at least 100 times than that of making a transition. In order to gain any advantage from the spatial and temporal locality of reference of the reordered states, and thus amortize the cost of reordering, the hot-spot would have to be exercised much more frequently than was done by the strings of length $4n - 4$.

A further experiment was therefore conducted to probe the best case scenario—one in which reordered states are traversed in the same order as they were generated. This was done by essentially repeating the previous experiment with the following modifications: strings were now of length $2n - 2$ instead of $4n - 4$; it was ensured that when the n^{th} symbol was encountered then the FA would be in state 0; and only the time taken to process the second group of $n - 1$ symbols was measured. Gnuplot was used to plot the graphs of number of states against time for both TD and SR algorithms. The results are provided in figure 4.

The graph shows that the TD processing time is super-linear in the size of the automaton. Although the SR trend appears to be close to linear, there is, in fact, a slight suggestion of superlinearity here as well. It is clear that in this case, SR enjoys

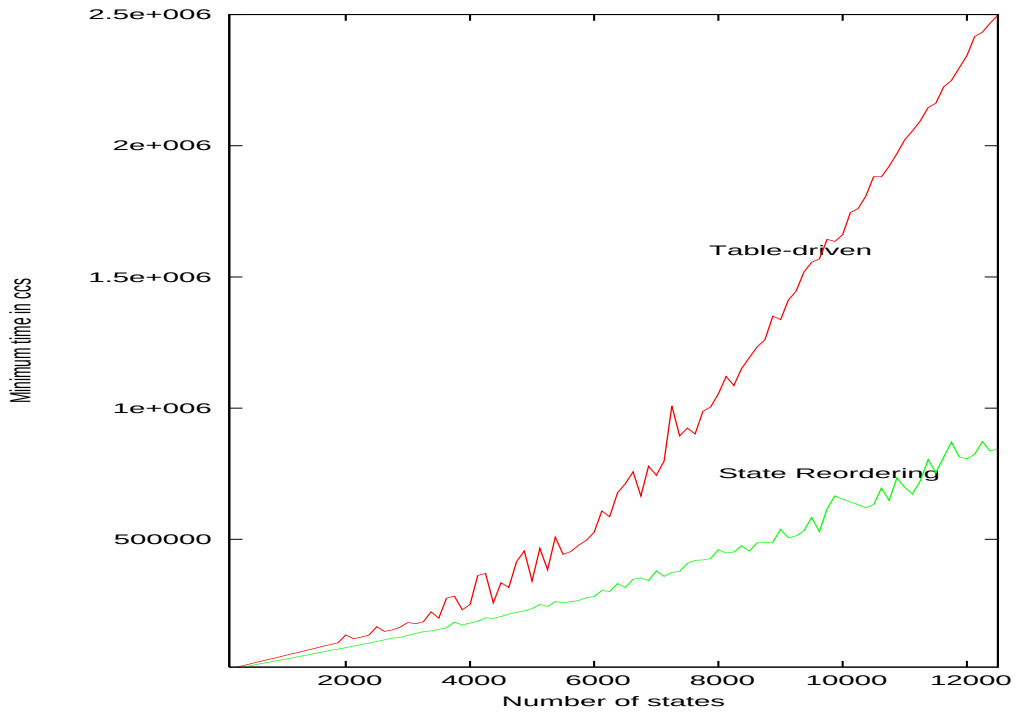

Figure 4: SR and TD Performance: Input String is Two Repeated Segments of Length $n-1$. Time is for Second Segment Only

a definite time advantage over TD, due to optimal cache utilization. For example, at about 9000 state transitions, the SR algorithm is about 60% faster than the TD algorithm. The graph may be thought of as SR's best case asymptotic behaviour. On this evidence, therefore, state reordering is a feasible strategy under conditions that approximate those discussed in section 3, item 3. Given that more recent hardware platforms have been placing increasing emphasis on additional cache memory[3], the gains obtained by optimally exercising cache are likely to increase.

5 Conclusion and Future Work:

In this paper, we have discussed the design of an algorithm for FA string recognition that attempts to leverage an advantage from the fact that cache memory relies on the principle of spatial and temporal locality of reference. Our experiments have suggested that the SR algorithm could gain an advantage over the traditional TD algorithm for long string sequences that tend to revisit hot-spot states in a certain order.

Two application areas that immediately suggests themselves as potentially worth exploring are DNA analysis and network intrusion detection. In the first case, one of the contemporary challenges is the identification of so-called micro- and/or mini-satellites (generically called approximate tandem repeats) in DNA strings. Here, what is sought is repeated approximate patterns in the string. The notion of "repetition"

[3]For example, the L2 cache of Intel's Prescott-2M Pentium 4 chip, released in February 2005, has 2048kB, while the Intel Intanium 2 processor, targeted for release in November 2005, will have a 3MB of L3 cache.

intuitively corresponds to the idea of processing within a hot-spot, as discussed earlier. In the latter case, one would imagine that scanning a stream of network data for security breaches involves, for the most part, the traversal of hot-spot states that should quickly pass the data down the line. Again, this seems like a possible application domain for the SR algorithm. However, a fuller investigation of appropriate application domains for SR is a matter left for future research.

The algorithm was implemented in NASM on an Intel Pentium 4 machine. Intel offers many data prefetching instructions for performance enhancement [6] that have not been used in the algorithm. These instructions should be analyzed in the future in the hope of speeding up even further the SR implementation.

The algorithm suggested in this paper is part of a set of algorithms under investigation. The aim is to package these in a dynamic framework for implementing FAs with a view to enhancing performance [4]. We are currently investigating a mixed-mode implementation as an alternative to the SR and hardcoded implementations explored to date. Once all the algorithms under investigation have been tested, the final design of the dynamic framework will be proposed as well as a toolkit for efficiently processing FAs.

References

[1] Ketcha Ngassam, E. *Hardcoding Finite Automata.* MSC Dissertation. University of Pretoria, 2003.

[2] Ketcha Ngassam, E., Watson, B. W. and Kourie, D.G. *Preliminary Experiments in Hardcoding Finite Automata.* Poster paper, CIAA, Santa Barbara, 299–300, September 2003.

[3] Ketcha Ngassam, E., Watson, B. W. and Kourie, D.G. *Hardcoding Finite State Automata Processing.* SAICSIT, Johannesburg, 111–121, September 2003.

[4] Ketcha Ngassam, E., Watson, B. W. and Kourie, D.G. *A Framework for the Dynamic Implemenation of Finite Automata for Performance Enhancement.* Prague Stringology Conference, Prague, August 2004.

[5] Hannessy, J. L., Patterson D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 3rd Edition, 2003.

[6] Intel Corporation. *The Intel Opimization Reference Manual.* http://www.intel.com/design/pentiumiii/manuals/, 2002.

[7] Thompson, K. *Regular Expression Search Algorithm.* Communications of the ACM. Volume 11, No 6, 323–350, 1968.

[8] Knuth, D.E., Morris Jr., J.H. and Pratt, V. R *Fast Pattern Matching in Strings.* SIAM J. Comput. Volume 6, No 1, 323–350, 1977.

[9] Yao, A C. *The Complexity of Pattern Matching for a Random String.* SIAM J. COmput., 8(3),pp. 368-387, 1979.

[10] Gerber, R., *The Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture*. Intel Corporation, 2002.