

# Hardcoding Finite State Automata Processing

E. KETCHA NGASSAM, BRUCE W. WATSON AND DERRICK G. KOURIE

Department of Computer Science University of Pretoria

URL: <http://fastar.cs.up.ac.za>

---

To hardcode and algorithm means to build into it the data that it requires. In this paper, we present various experiments in hardcoding the transition table of a finite state machine directly into string-recognizing code. Experiments are carried out in two phases. The first phase is limited to the analysis of the hardcoded behavior in relation to acceptance or rejection of a single symbol in some arbitrary state of some finite automaton. Then follows a simulation of the analysis of some hardcoded solution for recognizing an entire string. Measurements are provided to show the time efficiency gains by various hardcoded versions over the traditional table-driven approach.

Categories and Subject Descriptors: F.1.1 [**Theory of Computation**]: Computation by Abstract Devices—*Automata*; F.4.1 [**Theory of Computation**]: Mathematical Logic —*Computability Theory*; F.4.2 [**Theory of Computation**]: Grammars and their rewriting systems —*Grammar types, Parsing*

General Terms: Algorithms, Experimentation, Performance.

Additional Key Words and Phrases: Hardcoding, Automata, Pattern Matching.

---

## 1. INTRODUCTION

To hardcode an algorithm means to build into it the specific data that it requires. Even though this means that the algorithm cannot run with alternative data, a hardcoded algorithm may sometimes be more efficient than its softcode counterpart.

Implementers of finite state automata (FSAs) often use a table to represent the transition function. The conventional table-driven algorithm to determine whether an FSA recognizes a given string is generic in the sense that the transition table is part of the input data to the algorithm. The time taken by such an algorithm to determine whether an FSA recognizes a string, thus depends inter-alia on the memory load as represented by the size of the transition matrix. When manipulating large automata, the implementer has to be aware of, and indeed avoid, unpredictable behavior such as early program termination caused by memory overflow. This can be done by applying more complex techniques, such as vectorization [Dongarra et al. 1984] or recursion algorithms for efficient cache memory usage [Douglas et al. 1999; Bjarne et al. 1999].

Much of the work that has been done to improve automata implementation efficiency, has been at the modelling level; that is, before the automaton's transition table has been set up for processing by the standard algorithms. Automata minimization Watson[2002; 1995], and the study of specialized algorithms on problems using an FSA as basic model (Pattern Matching, Parsing, DNA analysis etc...) are among several examples available where the model is optimized before implementation.

Hardcoding automata has already proven to be effective in particular automata implementation areas such as parsing [Pennello. 1986; Horspool et al. 1988], and code generation [Achyutram et al. 1995; Fraser et al. 1991]. However, little has been said about hardcoding automata in a more general sense. This work helps to fill the gap. It consists of a performance analysis of various versions of string recognition algorithms based on hardcoded FSA implementation. However, instead of directly analyzing a hardcoded solution for recognizing an entire string, the analysis is first of all limited at the most elementary form possible: the analysis of hardcoded behavior in relation to acceptance or rejection of a single symbol in some arbitrary state of some finite automaton. This is followed by a simulation of the analysis of some hardcoded solution for recognizing an entire string based on the results obtained from previous experiments. We perform various tests using various coding strategies to help capture

---

Author Addresses:

E. Ketcha Ngassam, Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa; [eketcha@cs.up.ac.za](mailto:eketcha@cs.up.ac.za)

Bruce W. Watson, Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa; [bwatson@cs.up.ac.za](mailto:bwatson@cs.up.ac.za)

Derrick G. Kourie, Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa; [dkourie@cs.up.ac.za](mailto:dkourie@cs.up.ac.za)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, that the copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than SAICSIT or the ACM must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2003 SAICSIT

the advantage of using hardcode for situations where speed is a major factor. A conclusion of this work is that hardcoding may sometimes yield significant efficiency improvements over the traditional table-driven method, and might therefore be appropriate in particular circumstances where timing is important.

Section 2 presents the foundational background to the hardcoding experiments that were carried out. Section 3 explains the various approaches taken with a view to exploring optimization. Two levels of experiments were carried out, the first being where the size of the FA and the associated hardcode code was limited to an absolute minimum. Section 4 describes the data generation and collection processes used in this first round of experiments and section 5 presents a summary of these results. Section 6 describes a further experiment (the second level) induced by these previous results. Here various levels of cache and RAM memory is exercised, based on larger FAs than in the first experiments but relying on the best hardcoded algorithm previously encountered. Section 7 gives the overall conclusion and indicates directions for future research.

## 2. FOUNDATIONS

This section reviews a standard table-driven string recognition algorithm. It justifies the decision to restrict the study to an investigation of an automaton recognizing a single symbol and indicates in pseudo code how softcoding and hardcoding for such recognition should take place. It then reviews specific software support structures used for time measurement and random number generation.

### 2.1 The table-driven string recognition algorithm

A table-driven algorithm is the usual basis for ascertaining whether a string *str* is a member of the language generated by an FSA, *M*. Consider a string, *str*, of length *len* > 0, and a table *transition*[*i*][*j*] that represents the transition function of *M*, where  $0 \leq i < \text{numberOfStates}$  and  $0 \leq j < \text{alphabetSize}^1$ . If the automaton is in state *i* and the next character in the input string is *j*, then *transition*[*i*][*j*] indicates the next state of the automaton. Interpret *transition*[*i*][*j*] = -1 to mean that the automaton cannot make a transition in state *i* on input *j*. Algorithm 1 shows how the string *str* is conventionally tested for membership of the language accepted by *M* [Aho et al. 1986].

ALGORITHM 1. *Table-driven string recognition*

```

function recognize(str,transition):boolean
  state := 0;
  stringPos := 0;
  while(stringPos < len) ∧ (state ≥ 0) do
    state := transition[state][str[stringPos]];
    stringPos := stringPos+1;
  end while
  if state ≤ 0
    return(false);
  else
    return(true);
  end if
end function

```

### 2.2 Single symbol recognition

In the foregoing algorithm, each iteration of the loop processes a single symbol of an input string. Under normal circumstances, the time taken to process such a single symbol is independent of both the symbol itself and of its position in the string. Hence, a consideration of processing a single symbol instead of an entire string was assumed to be a reasonable basis for comparing various string processing algorithms. This does not imply that the study of each algorithm would be limited to a study of how it behaves in a single fixed state. Instead, both the symbol to be processed, and the transitions allowed from an assumed single state were randomly generated over many different simulation runs. In each case, a single state of an automaton has a randomly determined set of legal transitions on some set of randomly selected alphabet symbols and has no transition at all the remaining alphabet symbols.

Figure 1 shows the possible transitions associated with such single state of an automaton. The figure depicts a one-dimensional array, that may be regarded as a row of a transition matrix of some FSA. The array is indexed

<sup>1</sup>If the table is very sparse, its representation can also be based on linked lists, rather than on arrays. Details are not important in the present context, and do not materially affect the argument.

a	b	c	d	e	f	g	h
8	-1	-1	-1	10	14	5	5

Figure 1. A transition array for a state of some automaton

by the FSA's alphabet:  $\{a, b, c, d, e, f, g, h\}^2$ . Each entry of the table contains either a valid transition value (which indicates some arbitrary next state) or no transition at all (represented by the value  $-1$ ). Algorithm 2 depicts in pseudo code how the array can be referenced to determine whether an arbitrary input symbol is rejected in the given state of the automaton or not.

ALGORITHM 2. *Character testing on a state of some automaton*

```
function recChar(character,transition):boolean
    return(transition[character]≠-1);
end function
```

The point about this very simple pseudo code is that it is independent of the actual content of the transition array, in the sense that the pseudo code does not need to change for different values in the transition array. It is thus decidedly a specification for a softcoded version of the task at hand –it may be regarded as the version of table-driven algorithm 1 that has been trimmed down to dealing with one symbol in some arbitrary state described by the transition array.

### 2.3 Hardcoding the recognition of a single symbol

Hardcoding the recognition of a single symbol avoids the use of an array as a parameter, but generates instead, code that is specifically characteristic of a given transition array. Algorithm 3 shows an example, in pseudo-code, of the hardcode to deal with the single state transition array depicted in figure 1. Note that this hardcode has to change whenever a transition table with different entries is to be used.

ALGORITHM 3. *Hardcode of a transition array of some finite automaton*

```
function hardCode(character):boolean
if (character=b)∨(character=c)∨(character=d)
    return(false);
else if (character=a)∨(character=e)∨(character=f)∨
        (character=g)∨(character=h)
    return(true);
end if
end function
```

We are interested in determining how algorithms 2 and 3 compare in terms of their time performance. Of course, there are many sources of variation that require study in order to draw general conclusions:

- Algorithms 2 and 3 are presented in Pascal like pseudo-code language format. For empirical cross-comparison, they have to be translated into various high and low-level language implementations. Six different implementations were examined in this study (one of algorithm 2 and five of algorithm 3). These are described in section 3.
- The algorithms' performance may vary from one hardware platform to another. Only one hardware platform has been used in this study, namely an Intel Pentium IV processor.
- The algorithms' performance should be compared in relation to many different input scenarios. Algorithm 4 in section 2.5 describes how various randomized transition arrays were generated for the study.

### 2.4 Software support structures

The experiment involving cross-comparison of times taken for different algorithms demanded the use of software instructions to measure time. On Intel microprocessors, the so called time stamp counter keeps an accurate count on every cycle that occurs in the processor. The time stamp counter is a 64 bit model specific register (MSR) which is incremented at every clock cycle[Intel 2002]. Whether invoked directly as a low-level assembly instruction, or via some high-level language instruction, the RDTSC instruction allows one to read the time stamp counter[Intel 2002], and thus to determine approximately the time taken to execute critical sections of code. It was extensively used in the coded experiments of this study.

<sup>2</sup>For simplicity, we assume that alphabet symbols are permitted as array indices, and that indices are ordered in some reasonable way, e.g. in ASCII representation order.

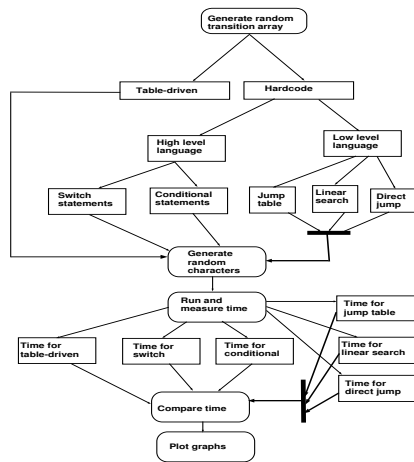


Figure 2. Process diagram indicating how the hardcoded implementation where compared to the table-driven implementation

The experiment also critically depended on generating randomized transition arrays such as depicted in figure 1. The function *ran2*, described in [Press et al. 2002], was selected for its efficiency for random numbers generated in a single calculation.

### 3. EXPERIMENTAL DESIGN

Figure 2 gives an overview of the main processes involved in the design and execution of the experiment. The process started with designing and implementing code that produces a random transition array. Such an array is used as input to a table-driven approach for recognizing a character of a string (i.e. an implementation of algorithm 2). It is also used to hardcode (i.e. implement algorithm 3) in several different ways. The figure shows two branches taken in regard to the hardcoding endeavor. The first provides two high-level language hardcoded implementations, and the second, three low-level (i.e. assembly) language hardcoded implementations. Figure 2 thus alludes to a total of five separate hardcode generating programs. Each such program takes as input, some transition array indicating how transitions from a single state of an FSA are to be made, and generates as output, the corresponding hardcode to handle a single input character in terms of such a transition array. In addition, a C++ version of the “table-driven” algorithm 2 also takes the same transition array as input, as well as the single randomly generated input character.

The entry marked “Generate random characters” in figure 2 therefore alludes to the generation of this random character that is used as input for a total of six different programs. As suggested in the figure, readings of the time stamp counter were appropriately embedded in each of the programs, making it possible to plot and compare the relevant execution times taken by each program.

#### 3.1 The Random Transition Array

For the purposes of the experiment, it was necessary to generate randomized transition arrays as shown in Algorithm 4.

ALGORITHM 4. *Generation of random transition array*

```

function genRandomArray(str):transitionArray
  i := 0;
  alphabetSize := ran2(0,255);
  numberOfTransitions := ran2(0,alphabetSize);
  {Initialization: transition[i] := -1
   for i ∈ [0..alphabetSize]}
  while (i < numberOfTransitions) do
    index := ran2(0,alphabetSize);
    transition[index] := ran2(0,maxInt);
    i := i + 1;
  end while
  return(transition);
end function
  
```

Such an array is based on the following random selections:

- the alphabet size of the FSA, say *alphabetSize*, where  $0 < \textit{alphabetSize} \leq 255$ ; and
- the maximum number of transitions,  $\textit{numberOfTransitions} \leq \textit{alphabetSize}$ , that can be made from the simulated single state.

The array is populated with random entries, each of which represents the next fictitious state to which a transition would be made if a full transition matrix were to be generated. The actual number of such fictitious states is largely irrelevant, but is represented by *maxInt*.

The array index position of each such entry is some random number  $< \textit{alphabetSize}$ . Such a random number is generated *numberOfTransitions* times. No provision is made to enforce the selection of a different random number with each generation, which means that sometimes an array index position may be reselected. Each array index position could be associated a symbol that is to be recognized, but which, for the purpose of the experiment, is merely taken to be the integer number itself.

The foregoing implies that an entry such as *transition*[12] := 234 should be interpreted to mean that if, in the current state, symbol 12 is encountered, then a transition should be made to the fictitious state 234. Symbols such as 12 are regarded as “accepting” symbols.

The implication of the above is that  $\textit{alphabetSize} - \textit{numberOfTransitions}$  is the minimum number of randomly located indices of the transition array that are not regarded as those triggering a transition. Instead these indices are associated with “rejecting” symbols.

### 3.2 The high-level experiments

In the case of high-level implementations, algorithm 3 was implemented in two ways. The first relied on the use of the switch statement, and the second, on the use of nested conditional statements (i.e. statements of the form: if...then...else...if...). The programming language chosen was C++, because of the ability of the g++ compiler to offer many optimization options (-O3). Linux was the OS of our choice. By way of example, short illustrative code extracts used in the two cases are provided in source-code 1 and 2 below.

SOURCE-CODE 1. *C++ extract for switch statement implementation*

```
...
switch(a)
{
case 0: nextState = 8; break;
case 5: nextState = 10; break
...
default: nextState = -1; break;
}
...
```

SOURCE-CODE 2. *C++ extract for nested conditional statement implementation*

```
...
...
if (a == 0) nextState = 8;
else if (a == 5) nextState = 10;
...
else nextState = -1;
...
...
```

### 3.3 The low-level experiments

Because of its simplicity, the Netwide Assembler (NASM)[Paul A. Carter.. 2002] was chosen to be used in implementing the various versions of algorithm 3.

The first version relied on a jump table. This approach was suggested by the disassembly of the high-level language’s switch statement. The latter revealed that the optimizing compiler constructs a jump table when encountering a switch statement with a large number of cases. However the compiler-generated jump table contained several generic features that could be further optimized for the specific scenario under study. A code extract that illustrates a jump table is shown in source-code 3. A table is defined in the data segment. Each table entry contains the start address of a block of code that is associated with an input symbol. This entry is accessed

by having regard to the table's start address and the value of the input symbol. The table entry relating to the input symbol is thus read, and the flow of control is directed to the memory containing instructions relating to that specific symbol.

SOURCE-CODE 3. *Sample code for jump table implementation in NASM*

```
segment      .data
;define the TABLE entries
TABLE      dd  case_default
           dd  case_0

...
...loop on alphabet size
;jumps begin here
    mov eax, [ebp-4]    ; store the value of the symbol in eax
    shl eax, 2         ; Byte offset from start of TABLE
    mov esi, TABLE    ; load start of TABLE address
    add esi, eax        ; determine the address of the jump
    jmp [esi]          ; jump to address
; specify the different case statements
case_default:
    mov edx, -1
    jmp next
case_0:
    mov edx, 8
    jmp next
...
```

The second assembler program carried out a sequential comparison of the various cases. Its structure is fairly obvious and is suggested by the disassembly of the high-level conditional statements (source-code 4). The implementation is essentially a linear search and is straightforward. It involves a sequential series of comparisons and jumps to the appropriate label, according to the current symbol.

SOURCE-CODE 4. *Sample code for linear search implementation in NASM*

```
...
    mov eax, [ebp-4]    ; store the value of the symbol in eax
    cmp eax, 0
    je near case_0
    cmp eax, 4
...
    je near case_default
; specify the different case statements
case_0:
    mov edx, 8
    jmp next
...
case_default:
    mov edx, -1
    jmp next
...
```

Yet another implementation strategy was investigated. It was based on the use of direct jumps in assembly. Source-code 5 shows a code extract. This direct jump version was an attempt to improve upon the jump table version. It involves the labelling of each block of statements that deals with a given symbol and ensuring that blocks are separated from one another by a constant offset. This makes it possible to compute a direct jump address from the symbol value and block size.

SOURCE-CODE 5. *Sample code for direct jump implementation in NASM*

```
;Direct jump using 10 as the size of each block of case statement
;
           and 7 as the size of the jump instruction itself
```

```

    mov eax, [ebp-4]    ; store the value of the symbol in eax
    mov edx, 10        ; store the size of the block in edx
    mul edx,
    add eax, \$.+7     ; eax = ip + 7 + 10*eax (absolute address to jump to)
    jmp eax           ; jump to address
; specify the different case statements
case_0:
    mov edx,8
    jmp next
case_1:
    mov edx, -1
    jmp next
...

```

#### 4. DATA COLLECTION

At the time of writing, 355 random transition arrays had been generated along the lines previously described. For each such array, the five versions of hardcode were generated. Each generated version was then compiled (high-level versions) or assembled. Then, running through each symbol of the alphabet (i.e. char in algorithm 2 and 3) each hardcode version as well as the code for table-driven implementation was run. The time taken to process each symbol was recorded, noting whether the symbol was an accepting or rejecting symbol of the state.

In order to account for statistical noise in the time variable (whether caused by the CPU or the Operating System), it was decided to repeat the measurements over 20 runs for the same transition array and input symbol.

Algorithm performance relative to accepting symbols, rejecting symbols, and all symbols collectively, was recorded in terms of three statistical measures, computed for each batch of 20 runs, namely: the average time, the minimum time, and the maximum time.

For the table-driven method as well as for each of the five hardcode versions this yielded a total of nine statistics per randomized transition array. In each of these cases a tenth statistic was also recorded, namely the randomly determined *number of accepting symbols* in the transition array. Note that, while the rejecting symbols are all treated similarly in the code, it is generally the case that the greater the number of accepting states, the greater the number of lines of code have to be added into the hardcoded versions to deal with these accepting states. In this sense, this tenth statistic, the average number of accepting states, will be regarded as a metric (albeit somewhat rough and ready) of the problem size.

The raw data thus consisted of six 355 by 10 matrices —one such matrix per each hardcoded version and another for the table-driven method.

In order to further smooth out statistical noise, it was decided to aggregate the 355 rows in each of the six above mentioned matrices. As a first step, the rows were sorted in ascending order of *number of accepting symbols*. The latter statistic was used as a basis for clustering contiguous rows, resulting in 33 clusters of rows, containing on approximately 10 to 11 rows per cluster. These clusters of contiguous rows were averaged by column. The net effect was to reduce each of the six matrices to 33 rows by 10 columns.

It should be noted that on some, rather rare occasions, the data contained outlier values. It seemed reasonable to assume that these measurements were so dominated by operating system or CPU chance events, that they merely obscured any legitimate conclusions that one might potentially draw about hardcode performance. Since practically all timing measurements were well within a 2-digit range, it was decided to regard any timing number greater than two digits as an outlier. These numbers were simply ignored in all subsequent computations.

#### 5. RESULTS

Various graphs can be constructed in which one of the nine time statistics from one of the six matrices is represented on the y-axis and the “problem size” (i.e. the tenth column of that matrix, the average number of accepting symbols), is on the x-axis. Thus, one could visually depict the impact of the problem size on, say, the average time taken to process a rejecting symbol when executing the *switch statement*; or alternatively on the minimum time to process an arbitrary symbol when executing the *table-driven* algorithm, etc.

Figure 3 gives three such graphs. These graphs depict, for each of the three hardcoded assembler versions, the impact of the “problem size” on the minimum time taken to process a rejecting symbol.

The following clearly stands out in the graphs.

—The problem size does not appear to have any impact on the speed of either the *direct jump* or the *jump table* hardcoded versions. The former turns out to be consistently more than twice as slow as the latter.

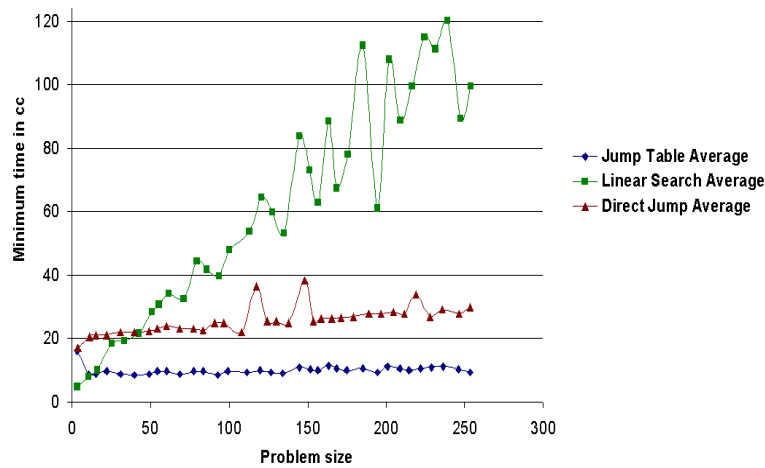


Figure 3. Performance based on rejecting symbols

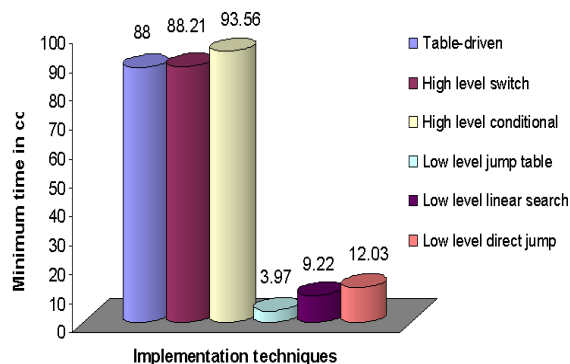


Figure 4. Average processing speed per implementation technique

- The *linear search* version displays a clear linear relationship to time, becoming increasingly slow as the problem size increases. This can of course be explained by the fact that the method requires that the input symbol be compared against each possible accepting symbol, before executing the code for a rejecting symbol.
- While the *linear search* version is slightly competitive with the *jump table* version for very small problem sizes, and with the *direct jump* for slightly larger problems, after a problem size of about 20, it becomes worse than both, and soon becomes hopelessly slow for very large problems.

Investigation of the rest of the data revealed that, with the exception of the graphs characterizing the *linear search* version, all of the graphs have more or less the same form as the *jump table* and *direct jump* graphs —i.e. they are substantially unrelated to problem size. The observation holds, irrespective of whether maxima, minima or average values are considered, or whether data relating to accept or reject symbols are used. As a result, it seemed reasonable to base further comparisons of the six different coding possibilities on average values taken over all problem sizes. The results are displayed in histogram form in figure 4. The figure relates to overall average performance (as opposed to overall minimum or maximum performance) in regard to any symbol (whether accepting or rejecting).

A similar overall picture is obtained if other metrics are used. In each case, the *jump table* version is more than twice as fast as its nearest rival (the *linear search* version) and more than forty times faster than any of the high-level implementation versions, whether table-driven or hardcoded. Globally speaking, hardcoding implementations in assembler are faster than other methods. Moreover, it would seem that hardcoding to a high-level language is not worthwhile, since the standard table-driven implementation seems to be slightly faster.

## 6. EXERCISING MEMORY LEVELS

The implementation of a hardcoded *string* recognizer will generally require a larger FA than generated above and will probably involve various levels of cache and RAM memory and possibly even paging. As a preliminary analysis of these matters, we have chosen to carry out an experiment using a very simple string recognizer that

was based on results obtained from the single state case described above. Unlike the table-driven algorithm, the code size for a hardcoded program for string recognition is dependent on the automaton size, i.e on the number of states. Memory management is therefore of concern, as the program's processing speed is likely to be affected by its executable code size.

One of the most common strategies to optimally process programs on Intel Architecture is by mean of Branch Prediction Buffer (BPB)[Gerber. 2002]. The processor predicts using the BPB the most likely set of instructions to be executed and loads it into cache memory for fast processing. For a given cache hit<sup>3</sup>, if the code is small enough to fit into L1, a memory fetch is done in about two clock cycles. For a set of instructions whose size is greater than the L1 cache's size, the processor requests the use of L2 for processing and shares the code between the two cache levels. This leads to a drop in the processing speed, simply because the two caches do not perform at the same speed. L2 cache requires a memory fetch of about 6 clock cycles to execute approximately 256KB of instructions. The RAM will be required by the processor if the cache memory is too small to contain the entire code to be executed. In this case, the performance slows considerably as this external memory is far slower than cache.

The experiment was based on a language that has only two symbols in its alphabet (say  $a$  and  $b$ ). For any state of the FSA (except the final one), an input of  $a$  always triggers a transition to a next state but  $b$  does not, since it is considered as a rejecting symbol. Our finite automaton only had one accepting state, namely its final state. The only string accepted by such an automaton with  $n$  states is the string  $a...a$  (i.e a string with  $n - 1$  symbols). Therefore, testing the processing speed required to recognize the above string represents the worst case scenario in the sense that any string containing a  $b$  symbol would be rejected sooner. The hardcoded for such an FSA can be easily constructed by selecting one of the hardcoded algorithms previously described and then appropriately linking together the NASM source code of  $n$  replications of this hardcoded. For the present experiment, the best performer of the algorithms was chosen, namely the jump table.

We generated 300 different FSAs with number of states ranging from 10 to 3000. Under the same conditions as in the single state case, 20 runs were performed for each hardcoded program. In each case, the corresponding time stamp counter value time to process the maximum length string was recorded and the minimum, maximum and average values over the 20 runs was noted. As a lower limit for our comparison, we assumed that the previous results for the jump table algorithm represent the best results possible. The ideal, excluding the effects of different levels of memory, would be that a linear time scale-up relative to the length of the input string would be achieved. The processing speed for various hardcoded multiple state programs was recorded. We then scaled the respective processing speeds for each algorithm's execution to get the average speed per state for automaton of various sizes. The series of data produced was plotted and cross compared with the jump table hardcoded experiment. Figure 5 depicts the resulting graph. The results for hardcoding multiple states shows fairly smooth growth in certain regions, but then changes linearly to another plateau. The overall pattern is plausibly explained by the following:

- The size of the L1 cache (8KB) cannot contain the smallest automaton (10 states) with its corresponding executable which is of the order of 17.9 KB. The program size is two times bigger than the L1 cache's size. Consequently, services of the L2 cache (256KB) are required for complete processing. This explains the fact that noise is indicated in the figure between 10 and about 480 states. The use of both L1 and L2 cache by the processor for branch prediction buffering increases the probability of cache misses at level L1, and cache hits at level L2. In other words, since few instructions are present in the L1 cache, they are frequently not needed whereas the needed part of the code is in the L2.
- A slow growth in time per state is experienced from about 480 states onwards. In the figure, between about 490 and 1610 states, the corresponding hardcoded executable appears to be able to fit into the L2 cache. This is illustrated by the smoothness of the curve within that range. As the number of states increases, the probability of having to reach beyond the L2 cache increases linearly, and hence there is also a slow but linear growth in the curve.
- A linear and rapid growth occurs between 2090 and 2570. The code to be executed apparently no longer fits into the L2 cache. Services of the RAM are required by the processor. Since the RAM is much slower than the cache memory and since the amount of RAM needed also increases linearly, the linear increase in average time per state seen in the figure conforms to expectation.
- The processing becomes stable after 2540 states. At this stage, it would seem that most of the hits are in the main memory. The overall processing speed of RAM explains such behaviour. We may expect that in the long

<sup>3</sup>A successful prediction by the processor is called a cache hit, otherwise it is said to be a cache miss

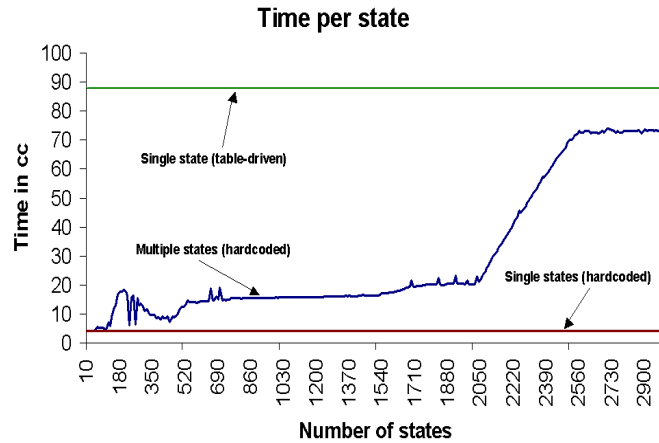


Figure 5. Hardcoded multiple state against single state implementation

run, if the memory is full, paging between memory and hard drive must be carried out. However, the effects of paging were not further investigated in this study.

As a final observation, note that the figure clearly indicates that, in the range of problem size currently investigated and based on the result obtained at this stage, hardcoding the recognition of a string appears to be more efficient than its table-driven counterpart. This is evident from the fact that the entire curve for the hardcode experiment lies below the horizontal line representing the time required per state for the table driven approach that was recorded in the previous experiment. However, a more complete investigation is needed of the cache misses and hits that would occur for a larger transition table when implementing table-driven approach. Such an evaluation would provide a realistic quantitative comparison between the two methods. It should be emphasized that the constant time per state shown in the figure for table-driven approach is *lower bound* on what would practically be experienced for transition tables of larger size.

## 7. CONCLUSION

This study firstly cautions against a naïve attempt to use a high-level language as a basis for hardcoding. It is perhaps not too surprising, however, that it affirms that hardcoding in assembler can lead to significant performance improvements in relation to the conventional table-driven implementation. It should be noted that, while the use of the *jump table* outperforms the current version of assembler code implementing the *direct jump*, there may be scope to reduce the time taken by the latter, based on further fine-tuning to optimize the type and number of arithmetic operations.

Nevertheless, the experiment carried out with the jump table extended to cover several states, strongly suggests that hardcoding will produce time efficiencies in comparison to the table-driven approach for a certain range of FSAs. The future challenge is to explore that range more fully. This will involve randomly generating more complex FSAs, investigating the cache effects on the table-driven algorithm and determining the bounds at which paging occurs for both hardcoded and table-driven versions.

## REFERENCES

- THOMAS J. PENNELLO. Very Fast LR Parser. *Proceedings of the SIGPLAN Symposium on Compiler Constructions 21*, 7 (July 1986.), 145–151.
- R. NIGEL HORSPOOL, and M. WHITNEY. Even Faster LR Parsing. *Software Practice and Experiences 20*, 6 (June 1988.), 515–531.
- ACHYUTRAM BHAMIDIPATY, and TODD A. PROEBSTING. Very fast YACC-Compatible Parsers (for Very Little Effort), Department of Computer Science university of Arizona, 1995.
- CHRISTOPHER W. FRASER, and ROBERT R. HENRY. Hard-Coding Bottom-up code generation tables to save time and space. *Software-Practice & Experiences 21*, 1(Jan. 1991), 1-12.
- BRUCE W. WATSON. Directly Constructing Minimal DFAs: Combining Two Algorithms by Brzozowski. *SART/SACJ*, No 29, 2002, 17-23
- BRUCE W. WATSON. Taxonomies and Toolkits of Regular Language Algorithms. PhD Thesis, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, September 1995.

- CRAIG C. DOUGLAS, JONATHAN HU, MOHAMED ISKANDARANI, MARKUS KOWARSCHIK, ULRICH RÜDE, AND CHRISTIAN WEISS. Maximizing Cache Memory Usage for Multigrid Algorithms. Proceedings of the International Workshop held at Beijing, China, August 2-6, 1999, Lecture Notes in Physics, pp. 124ff. Springer, August 2000.
- J. J. DONGARRA, F. G. GUSTAFSON, AND A. KARP. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. SIAM Review, 26 (1984), pp. 91-112.
- BJARNE S. ANDERSEN, ALEXANDER KARAIVANOV, JERZY WASNIEWSKI, FRED GUSTAVSON, AND PLAMEN Y. YALAMOV. Linear Algebra With Recursive Algorithms. <http://lawra.uni-c.dk/lawra/abstracts/KazDolny99/KazDolny99.html>
- INTEL CORPORATION. The Intel Architecture Optimization Reference Manual, 2002. <http://www.intel.com/design/pentiumiii/manuals/>
- WILLIAM H. PRESS, SAUL A. TEUKOLSKY, WILLIAM T. VETTERLING, AND BRIAN P. FLANNERY. Numerical recipes in C++: the art of scientific computing. Cambridge, UK; New York: Cambridge University Press, 2002.
- RICHARD GERBER. The Software Optimization Cookbook: High-performance Recipes for the Intel Architecture. Intel Corporation, 2002.
- E.KETCHA NGASSAM, BRUCE W.WATSON, DERRICK G. KOURIE. 2003. Preliminary Experiments on Hardcoding Finite Automata. 8th International Conference on Implementation and Application of Automata, CIAA, 2003. LNCS 2759, pageff 299. Springer, 2003.
- PAUL A. CARTER. PC Assembly Language: <http://www.drpaulcarter.com/pcasm>
- A.V. AHO, R. SETHI, AND J.D. ULLMAN. Compiler: Principles, techniques and tools. Addison-Wesley, 1986.