Chapter 1

Introduction

Performance considerations often form an important part of problems that require computational solutions. Sometimes solutions to problems need to be finely tuned to produce an outcome that results in near minimal timing. It might be easy to find a brute force solution to a given problem, but it is usually harder to elaborate an optimal solution. Over the years, numerous techniques have been developed to improve performance, no matter the computational domain of concern.

This dissertation focuses on the development of a technique aimed at improving performance of certain computational problems related to finite state automata (or simply finite automata and abbreviated to *FAs*). The overall objective is to explore the hardcoding of algorithms related to *FAs*, as an alternative to the traditional softcoded implementation approach to *FA*-related processing.

1.1 The Problem

To hardcode an algorithm means to build into it specific data that it requires. The algorithm is implemented in such a way that it will not require – and indeed cannot handle – alternative data at run time. The algorithm of concern in this dissertation is one that determines whether or not an arbitrary input string is an element of the language represented by an *FA*. Throughout the dissertation references to implementing

(or hardcoding) an *FA* should be construed to mean implementing (or hardcoding) such and algorithm. Hardcoding this algorithm therefore involves the use of primitive data types and very simple control structures to represent the entire automaton. The algorithm in this case is a set of simple instructions with embedded data, whereas its softcoded version requires the use of data in memory to represent the *FA*'s transition function. Of course, in both the hardcoded and softcoded algorithms, the arbitrary input string to be tested is not "hardcoded" in any sense, but is rather ordinary input that can change with each execution of the algorithm.

Implementers of *FAs* generally use a table to represent the transition function. The conventional table-driven algorithm to determine whether an *FA* recognizes a given string is generic in the sense that the transition table is part of the input data to the algorithm. At any stage a different *FA* can be considered, simply by changing the transition table. The time taken by such an algorithm to determine whether an *FA* recognizes a string, thus depends inter-alia on the memory load as represented by the size of the transition matrix. When manipulating very 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 [DoGK84] or recursion algorithms for efficient cache memory usage [Douglas et al 00, Andersen et al 99].

Much of the work that has been done to improve automata implementation efficiency, has been done at the modelling stage – that is, before the automaton's transition table has been set up for processing by the standard algorithms. Automata minimization [Wat02, Wat95], and the study (for improvement) of specialized algorithms on problems using *FA* as basic model such as pattern matching, parsing, DNA analysis, neural network etc are among several examples where the model is optimized before implementation.

Since much work to optimize the representation of *FAs* has already been done at

the modelling level, hardcoding *FA* algorithms, aimed at enhancing processing speed, seems like an inevitable next step. In this dissertation, therefore, we investigate whether hardcoded implementations of *FAs* may be more efficient than the traditional table-driven implementation of *FAs*.

1.2 *FAs* in Context

Chomsky classified formal languages in four basic categories as follows:

1. Right-linear languages;
2. Context Free Languages;
3. Context Sensitive Languages; and
4. Unrestricted or Self Recursive Languages.

Each of the above enumerated language has its equivalent machine that might be use for practical modelling and implementation. They are respectively, Finite automata (*FAs*), Push down automata (*PDAs*), Linear Bounded automata (*LBAs*), and Turing Machines (*TMs*). More importantly, the inclusion relation holds when applied to the above hierarchy from top-to-bottom – i.e. context free languages include all the right linear languages, context sensitive include the context free, etc. The present empirical study is restricted to the first element of the Chomsky hierarchy, namely to *right-linear grammars*. We do not however restrict ourselves to some particular right-linear grammar. Rather, the experiments performed relate to *FAs* in general.

It is not the intention to extend the study to other types of languages in the Chomsky hierarchy. In fact, as shall be seen in chapter 2, considerable work in the direction of this present study has already been carried out in regard to parsing strings from a context free language. In a sense, one could therefore say that the present work seeks to specialize existing results, some of which are already routinely

applied when parsing context free languages. However, such specialization is more at a logical level – the study does not take explicit cognizance of the of existing context free grammar techniques and specialize them for right linear grammars. Rather, it investigates solutions for right linear grammars *ab initio*, in a bottom-up fashion, as it were.

1.3 Objective of the dissertation

This works aims to show that any implementation of *FAs* where speed is major factor may gain from the results obtained. The longer term objective is to produce a toolkit for processing hardcoded *FAs*. But to achieve this, the work must be carried out in a stepwise fashion. Thus, it must first be established whether hardcoding a *FAs* would indeed be more efficient than a table-driven implementation. Having established that, more work will have to be done at a later stage in order to implement the actual toolkit. The work consists of a performance analysis of various versions of string recognition algorithms based on hardcoded *FA* implementations. It aims to examine whether and to what extent hardcoding *FAs* is more efficient than the table-driven method when speed is the major factor to be considered. It also investigates whether the implementation of *FAs* using high level languages is inefficient compared to their low level language implementation counterparts.

1.4 Methodology

The following approaches are used to achieve our goal:

- A theoretical presentation of both methods is provided in the form of generic pseudo-code that describes the essential algorithmic issues relevant to both cases.

- This provides a basis for a theoretical cross comparison of the methods prior to the empirical assessment of their efficiency,
- For the purposes of the empirical study, the problem is initially restricted to a highly constrained domain. However, it is argued that this is done without loss of generality in regard to the broad conclusions.
- The empirical study is then carried out on randomly generated *FAs* and randomly generated input symbols,
- The results are then presented, providing a quantitative comparison between hardcoding and table-driven with respect to time efficiency.
- A study of the cache memory effects is carried out, based on the implementation of string recognition algorithms by *FAs* using both hardcoded and table-driven methods.

A conclusion of this work is that hardcoding may sometimes yields significant efficiency improvements over the traditional table-driven method, and might therefore be appropriate in particular circumstances where timing is important.

1.5 Dissertation Outline

Chapter 2 provides mathematical preliminaries as background to the remainder of the work. It also reviews a number of conventional computational concepts that use *FAs* as a basic model. Two levels of experiments were carried out, the first being where the size of the *FA* and the associated hardcode was limited to an absolute minimum. Chapter 3 discusses the design of these first level experiment and shows their relevance. Chapter 4 presents tools and methodology required to carry out the first level experiments. Chapter 5 depicts implementations details used in the first round

of experiments and the quantitative results are presented and discussed in chapter 6. Chapter 7 describes a further experiment induced by these previous results. Here, various levels of cache and main memory are exercised, based on larger *FAs* than in the first experiment but relying on the best hardcoded algorithm previously encountered. Chapter 8 gives the overall conclusion and indicates directions for further research.

Chapter 2

Background and Related Work

2.1 Introduction

This dissertation involves practical experimentation – specifically in regard to the time-efficiency implications of hardcoding a finite automaton. However, by way of introduction, it is appropriate to locate the theoretical setting of the broad problem domain. Hence, this chapter introduces theoretical aspect of *FAs* and their relationship to various well-known computational problems that rely on various other kinds of automata as their primary model for the solution. The chapter summarizes standard introductory texts such as [McN82], and [LePa81]. We formally define *FAs* and explore two major characteristics of such devices, namely *determinism and non-determinism*. This helps clarify the kinds of automata related to hardcoding algorithms that shall be encountered in the remainder of the dissertation.

Sections on regular expressions, patterns matching and context free grammars are presented. Emphasis is placed on their respective relationship to *FAs* in order to give an indication of the wider domain of applicability of finite state automata in the computing world. The chapter ends with an explicit presentation of what has been done in hardcoding finite automata, merely for improving syntactical analyzers.

Hence producing efficient LR parsers and one of its subsequent parser generators such as YACC[1].

2.2 Finite Automata

From a practical point of view, a finite automaton is considered to be a computing device used to examine (recognize) strings over an alphabet. The study of the complexity of such a device is of importance due to its wide area of applicability. The goal of its implementation is to accept or reject an input string. It is therefore of importance to consider how much memory the device uses for the processing of a given string, as well as its time complexity or processing speed.

An example of *FA* is shown in figure 2.1, and more explanations may be found in [LePa81]. The finite automaton has a transition function δ, where for each state s_i, and for zero or more characters c in the device's alphabet, the operation $\delta(s_i, c)$ maps to some state that depends on the value of c. Once it has read a character, the *FA* makes a transition $\delta(s_i, c)$ to some new state. The automaton halts when there are no more character to be read from the *input stream*, or when $\delta(s_i, c)$ in not defined.

When setting up or constructing an *FA*, a set of its states are designated as final states. The device is said to recognize a string if and only if there is a sequence of transitions of the device that leads to one such final state after reading the last character of the string. Otherwise, the device does not recognize the string. The characters read by the *FA* from the input stream are part of its alphabet. The set of all strings that an *FA* accepts is said to be the *language* recognized by the *FA*.

An *FA* may be modelled as a transition graph as shown in figure 2.2 as long as its *grammar* is well defined.

The *grammar* of a finite automaton is a *formal description of the structure of all*

[1]Yet Another Compiler Compiler

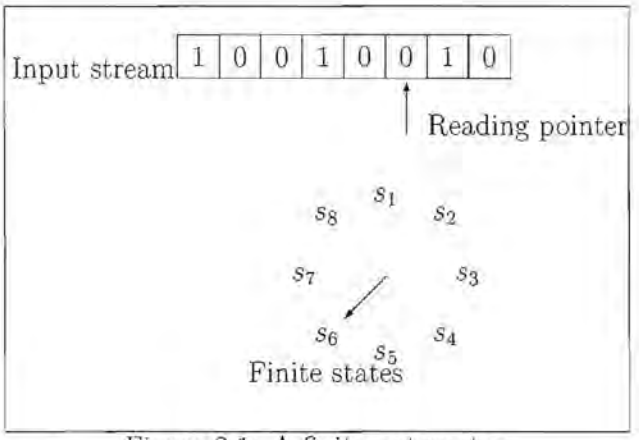

Figure 2.1: A finite automaton

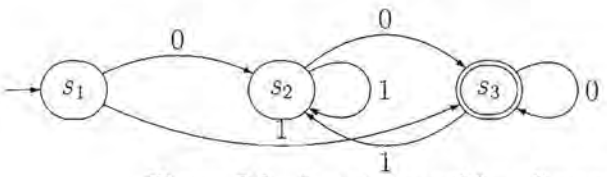

Figure 2.2: A state transition diagram

finite and allowable sequences of symbols that can be obtained using the alphabet. The transition function is part of such a formal description. We shall see later that a regular expression is an alternative way of formally describing such a grammar.

2.2.1 Deterministic Finite Automaton (DFA)

As defined in [McN82], a *deterministic finite automaton* M is a quintuple (S, A, δ, s_1, F) where:

- S is a finite set of states,
- A is an alphabet,
- $s_1 \in S$ is the initial state,
- $F \subseteq S$ is the set of final states, and

- the transition function $\delta : S\text{x}A \rightarrow S$, is defined such that

$$\forall s \in S, \forall c \in A, \delta(s,c) \in S \quad \text{is uniquely determined.}$$

2.2.2 Complexity of DFA String Recognition

The *complexity* of the computation of a string recognition operation by a deterministic finite automaton is measured in terms of complexity of each transition to be made (if any) for a given character in the string.

Assume that a string *str* is defined by $str \equiv c_1c_2...c_{n-1}c_n$, and assume that M, is a DFA as defined above.

We denote by $X_M(str)$ the *complexity* of recognizing the string *str*.

Thus

$$\begin{aligned} X_M(str) &= X_M(c_1c_2...c_{n-1}c_n) \\ &= X_M(c_1) + X_M(c_2) + ... + X_M(c_{n-1}) + X_M(c_n) \\ &= \sum_{i=1}^{n} X_M(c_i) \end{aligned}$$

where $X_M(c_i)$ denotes the complexity of recognizing the single character c_i. If we assume that the complexity of recognizing a single character can be characterized by some constant value, say K, irrespective of the state or the character, then both the worst case complexity and the average case complexity of recognizing a string may be regarded as linearly dependent on n, the length of the string – i.e. the complexity is $K \times n$. However, in practical implementations (as will be seen later) the time taken to implement the operation $\delta(s_j, c_i)$ is a statistical variable.

The transition function δ of a deterministic finite automaton is said to be *total* if and only if

$$\forall s_i \in S, \forall c \in A, \exists s_j \in S : \delta(s_i, c) = s_j \ (s_j \text{ is unique}).$$

Then, if a string *str* is recognized by the automaton M, it follows that $\delta(s_i, c_j) = s_k$,

with $i, k = 1, ..., m$, and $j = 1, 2..., n$; where m is the total number of states, and n the numbers of characters in str.

The transition function δ of a deterministic finite automaton is said to be *partial* if and only if

$$\exists s_i \in S, \exists c \in A : \delta(s, c) \text{ does not exist.}$$

If a string str is recognized by M, the complexity of each c_i exists and is totaly determined. That is, $X_M(str) = \sum_{i=1}^{n}(c_i)$.

If a string str is not recognized by M, then

$$\exists i, 1 \leq i \leq n : \delta(s_i, c_n) = s_k \in S - F, \text{ or } \delta(s_i, c_j) \text{ does not exists.}$$

In this second case, c_j is the first character of str, that does not allow a transition at the state s_k. This means that, for the present purposes it may be assumed that $X_M(c_l) = 0, j < l \leq n$. In particular, for $i = 1$, if $\delta(s_1, c_1)$ does not exists, then $X_M(str) = X_M(c_1)$.

2.2.3 Non-Deterministic Finite Automata (NFA)

A finite automaton is said to be *non-deterministic* when the transition operation on a given state and a symbol returns a set of states as result. A non-deterministic finite automaton (NFA) M is a quintuple (S, A, Δ, s_1, F) where,

- S is a finite set of states
- A is an alphabet,
- $s_1 \in S$ is the initial state,
- $F \subseteq S$ is the set of final states, and
- $\Delta \subseteq S \times A \times S$ is the transition relation. Thus, $(s_i, u, s_j) \in \Delta$ means that when in state s_i and presented with the input character u the automaton M

may transit to the state s_j. However, if it is also true that $(s_i, u, s_k) \in \triangle$, then when in state s_i and presented with the input character u the automaton M may transit to the state s_k. The particular state that is selected cannot be predetermined. It is in this sense that the *NFA* is non-deterministic.

There are various practical reasons for using non-deterministic models. In some cases, a non-deterministic model reflects the underlying random nature of that solution which is being modelled. In other cases, non-determinism is used in a specification context. The non-deterministic model concisely specifies several possible alternative options to be implemented, all alternatives being acceptable as implementations of some system. The implementer is then free to select any one of the nondeterministic possibilities that have been specified. Examples of *NFAs* may be found in [LePa81].

2.2.4 Equivalence DFA and NFA

As pointed above, two automata are considered equivalent if they accept the same language no matter what method is used to accept the language. Moreover, based on the definition of *NFAs*, it can easily be seen that any *DFA* can be regarded as a special case of a *NFA* – i.e. one that is restricted to having uniquely-defined transitions. The inverse can also be shown to be true, as enunciated in the following theorem.

Theorem

For each non-deterministic finite automaton, there is an equivalent deterministic finite automaton. The proof of this theorem will not be given here. It can be found, together with various illustrative examples, in [McN82].

2.3 Finite Automata and Regular Expressions

A regular expression (RE) is a formula for specifying strings that conform to some pattern [AhUl72]. It may be composed of characters (symbols or terminals) and meta-characters (non-terminals).

A regular expression is thus used to specify a pattern of strings. It is an algebraic formula consisting of a pattern whose value is set of strings. This set of strings is called the language of the regular expression.

A regular expression is defined over an *alphabet*, that is the set of symbols found in the derived strings and used in specifying the regular expression.

Just like any algebraic formula, a regular expression is composed of one or more *operands* appropriately conjoined by zero or more *operators*.

2.3.1 Operands and Operators of a Regular Expression

In a regular expression, operands can be:

- a symbol or terminal from the alphabet over which the regular expression is defined; or
- a non-terminal (group of symbols or variables) whose values are any pattern defined by the regular expression; or
- the empty string denoted ε; or
- the empty set of strings denoted ϕ.

Operators required to compose a regular expression include the following:

- Union: if R_1 and R_2 are two regular expressions, so is $R_1 \cup R_2$. Therefore, $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$. Where L represents the language defined by the regular expression.

- Concatenation: if R_1 and R_2 are two regular expressions, so is R_1R_2. Therefore, $L(R_1R_2) = L(R_1)L(R_2)$

- Kleene Closure: if R is a regular expression, then R^*(Kleene Closure of R) is a regular expression. And, $L(R) = \varepsilon \cup L(R) \cup L(RR) \cup L(RRR) \cup ...L(RRR....)$

An algebraic formula that constitutes a regular expression will therefore be any valid combination of *operands and operators* that have been defined above. For example, the set of strings over the alphabet $\{a, b\}$ that end with two consecutive b's has the following algebraic formula: $(a \cup b)^*bb$.

2.3.2 Equivalence of Finite Automata and Regular Expressions

From a regular expression, we can derive the related finite automaton and vise-versa. The following propositions relate regular expressions and finite automata.

1. For every regular expression R , there is a corresponding finite automaton M that accepts the set of string generated by R.

2. For any finite automaton M, there is a corresponding regular expression R that generates the set of strings accepted by M.

The proof of the propositions is can be found in [LePa81].

Corollary

A language is regular if and only if it is accepted by a finite automaton. Refer to [LePa81] for a constructive proof.

2.3.3 Summary of the Section

We have discussed in this section the notion of a regular expression and its relation to finite automata. Regular expressions are widely used in computing, especially for searching within a string for a substring that matches some pattern, where the pattern to be matched is expressed as a regular expression. The efficiency of the algorithm that implements such a matching is therefore of interest and our work lies in this category. The section below discusses pattern matching algorithms in more detail.

2.4 Pattern Matching

Pattern matching is the process of matching, completely or partially, some occurrence of a given pattern (string of characters) against substrings of text. The algorithm outputs a set of positions in the text where a substring starts that is a partial or an exact match of the pattern. In most cases, the pattern is a finite set of substrings within the text.

2.4.1 General Pattern Matching Algorithm

In the forgoing, *EOT* denotes a special marker for the end of a text (a string). In general, the algorithm for matching a pattern P in a text T may be expressed as follows:

Algorithm 1. *Simple matching algorithm*

function *simpleMatch(T: Text; P: Pattern):*ArrayOfMatches
  t,p := 0,0;
  results := $\emptyset$;
  while *(T[t]* $\neq$ *EOT)* **do**
    if *(T[t]* = *P[p]* $\wedge$ *P[p] = last character of P)* **then**

update ListOfMatches;

p := 0;

if *(T[t] = P[p] ∧ P[p] ≠ last character of P)* **then**

p := p + 1;

if *(T[t] ≠ P[p])***then**

p := 0;

t := t + 1;

end while

return results;

end function

The algorithm receives a text T and the pattern P and outputs *results* representing all occurrences of P in T. We can generalize the algorithm so that instead of looking for occurrences of a single pattern, we define a set of pattern to match against the text and modify the algorithm accordingly. The algorithm as presented above is for exact string matching; a modification can also be made to handle partial string matching problems. Various forms of pattern matching algorithms can be found in [Wat95], and [Cleo03]. Our problem is to present their relevance in general to *FAs*. The section below outlines the relationship between pattern matching problems and finite automata.

2.4.2 String Keyword Pattern Matching and Finite Automata

*A*ssume that *(PMs)* is a string keyword pattern matching problem in which some arbitrary set of patterns is to be recognized. A finite automaton can be constructed that recognizes all the string described in the set of patterns.

Proof

This proof can be given in two steps: firstly a single pattern to match against a text is considered; and secondly, the problem is generalized to a finite set of patterns to be match against a text. The details of this two-step proof may be found in [CroHa97].

2.4.3 Summary of the section

Various pattern matching problems can thus be solved by using appropriate *FAs*. To the extent that *FAs* can be successfully hardcoded, so too can pattern matching algorithms be hardcoded. This supports the case for experimenting with the hardcoding of *FA*-related algorithms. Such experimentation may ultimately result in processing speed improvements for various pattern matching problems.

One of the most popular application of *FAs* processing is that of compilers. Lexical analysis is an important phase in compiling and use *FAs* as basic computing model. The section below briefly summarizes lexical analysis.

2.5 Lexical Analysis

In compiling, lexical analysis is the process of converting the source code of a program into a stream of tokens. Some of the functions performed by the lexical analyzer include: removal of white space and comments, collecting digits into integers, and recognizing identifiers and keywords. Given a specification of a regular expression that defines lexemes[2] in the language, the compiler writer can directly construct an *FA* that recognizes inputs from the source code and performs various other tasks such as creating a symbol table. The process of constructing a lexical analyzer can be done directly by a code generator such as Lex [LeSc75]. Lex takes as input a

[2]The character sequence forming tokens

regular expression specification that represents the legitimate patterns of the lexical analyzer to be created, then generates a C program. The C code is then executed to produce an output that represents the lexical analyzer, which consists of the transition table and a "driver" that performs various operations such as scanning the input stream and producing a sequence of tokens according to the matching revealed by the transition table. Lex therefore performs a so-called table-driven generation of a lexical analyzer. However, it is possible to implement a hardcoded lexical analyzer as suggested in [AhSU86]. Our work involving an empirical study of hardcoding *FAs* in general, therefore relates to hardcoding lexical analyzers, since they merely model *FAs* and additionally perform various other ad-hoc functions.

2.5.1 Summary of the section

Lexical analysis summarized above is one of the key problems directly related to *FAs* implementation and therefore, may be hardcoded for efficiency. Many other computational problems such as graphic animation, image processing, artificial intelligence applications, cryptology and circuit design, protocol implementation, genomic, etc use *FAs* as a basic computational model. In many of these areas, the solutions to the problems encountered rely on the processing of automata (sometimes very large) using the classical table-driven approach. Hence there is a need to investigate efficient solutions by means of hardcoding as a possible alternative.

Yet another topic that relates to *FAs* is that of context free languages. The section below focuses on context free languages and their relation to finite automata.

2.6 Context Free Grammars

A context free grammar (*CFG*) is a formal system that describes a language by specifying how any legal expression can be derived from a distinguished symbol called

the axiom, or sentence of symbols [ELI02]. A *CFG* is normally specified as a set of recursive rewriting rules (or productions) used to generate patterns of strings. *CFGs* are often used to defined the syntax of programming languages. A CFG consists of the following components:

- A set of terminals symbols, which are the characters of the alphabet that appear in the strings generated by the grammar.
- A set of non-terminals symbols representing patterns of terminal symbols that can be generated by the non-terminal symbols
- A set of productions, representing rules for replacing non-terminals symbols on the left side of the production in a string with other non-terminal or terminal symbols on the right side of the production.
- A start symbol, which is a special non-terminal symbol that appears in the initial string generated by the grammar.

Formally, a *CFG* is a quadruple $G = (A, V, P, S)$ where:

- A is the alphabet of terminal symbols
- V is the set of non-terminals
- P is the set of productions (rules such that $P \subseteq V\text{x}(V \cup A)^*$)
- S is the start symbol, which is an element of V.

2.6.1 Definition

The notation $a \Rightarrow_G^* b$ for some non-terminal symbol a and some string b means that there exists a so-called derivation from a to b using various production rules P from

G. The set of strings consisting of terminal symbols only that can be derived from the start symbol of a CFG is a context free language (CFL).

Thus, if G is a context free grammar, then its corresponding context free language is $L(G) = \{w \in A^* : S \Rightarrow_G^* w\}$.

2.6.2 Context Free Grammars and Regular expressions

Context free grammars are more powerful than regular expressions (therefore finite automata) in the sense expressed by the following theorem:

Theorem

Any language that can be generated using regular expressions can be generated by a context free grammar. But there are languages generated by context free grammars that cannot be generated by any regular expression. The proof of this theorem can be found in [AhUl72].

As described in [AhUl72], strings in a CFL can be recognized by using an appropriate *push-down automata*, described in more details below.

2.6.3 Push Down Automata

A Push Down Automaton (PDA) is a finite state machine that is equipped with a memory device that functions as a *push-down-store*. $PDAs$ are accepting devices for context free languages. They are therefore equivalent to context free grammars in the sense that, given any context free grammar G, a push-down automaton A can be derived that recognizes just the sentences generated by G.

A PDA consists of four components:

- a control unit (CU),

- a Read unit (RU),
- an input stream (IT), and
- a memory unit (MU).

The control unit, the read unit and the input stream are the same as those of a FA, except that the transition executed by the control unit of a PDA involves operations to store symbols in, and retrieve symbols from its memory unit.

The memory unit of a PDA operates as a stack or push-down-store. As shown in figure 2.3, the stack of a PDA can be treated as a list; symbols can be added, remove only form one end of the list called *top* of the store. The push down store also

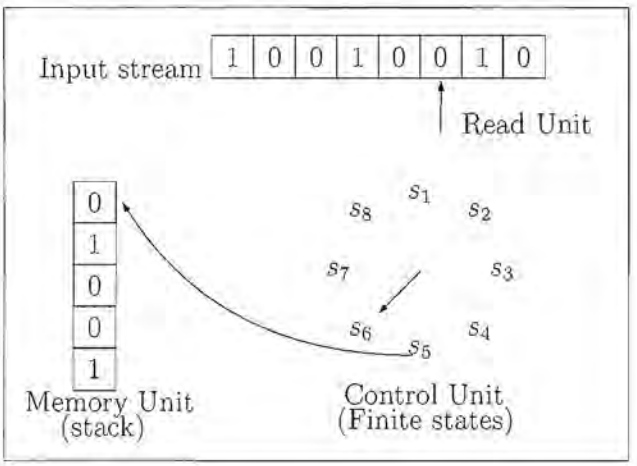

Figure 2.3: A Push Down Automaton

contains the *empty* pointer that enables to know if the store is empty or not. The store is said to be empty if $top = empty$.
a PDA is therefore a sextuple $(K, A, Z, \Delta, q_0, F)$ where:

- K is a finite set of states,
- A is a an alphabet representing the symbols of the input stream,

- Z is an alphabet representing the set of characters in the store,
- Δ is the transition relation of the machine which maps from

$$K \times A \times Z \longrightarrow Z^* \times K$$

- $q_0 \in K$ identifies the starting state of the machine, and
- $F \subseteq K$ is a set of final states.

Configuration

The configuration of a PDA at any stage in its processing of an input string is determined by its current state and the contents of its store. For example, consider a PDA currently in state q_i, where its push down store, in addition to the empty store symbol z_0, contains the symbols $z_1 z_2 ... z_k$ with z_k being the topmost. The configuration of the automaton is then specified by the string $z_0 z_1 z_2 ... z_k q_i$. In general, if the store contains the string $\eta \in Z^*$, then the configuration of the machine when in state q_i is $q_i z_0 \eta$.

Transitions function

The transition function Δ, of a PDA may be represented as a set of ordered quintuples $(q_i, a_i z_i, q_j, w)$ where;

- $q_i \in K$ is a label identifying the current state
- $a_i \in A$ is a symbol that can be read when the automaton is in the state q_i (if the automaton does not need to read a character in order to make a transition, then the symbol is the empty symbol ϵ)
- $z_i \in Z$ is the topmost symbol is the store

- $q_j \in K$ is the label of the next state
- $w \in Z$ is the word contained in the store in the state q_j.

Using the rewriting rule format, the above transition can be written as

$$q_i a_i z_i \longrightarrow q_j w \text{ or } (q_i, a_i, z_i) \longrightarrow (q_j, w) \ .$$

Accepting conditions

A PDA that has read all the symbols of an input string $\mu \in A^*$ and halts in the configuration $q_f z_0 \eta$ is said to accept μ if either one of the following conditions is met:

- *Final state condition:* the state q_f is an final state of the PDA – that is, $q_f \in F$, where F is the set of final states. The string symbol $\eta \in Z^*$ left in the store above the empty store symbol z_0 need not necessarily be null, although it can be the case that $\eta = \epsilon$.
- *Empty store condition:* the push down store is empty, that is, it contains only the empty store symbol z_0 meaning that $\eta = \epsilon$. The state q_h in which the machine halts need not necessarily be a final state, although it can be the case that $q_f \in F$.

Language of a PDA

The language $L(A)$ of a push down automaton A is defined as follows

$$L(A) = \{\mu : \mu \in A^*, q_0 \mu z_0 \Longrightarrow^* q_f z_0 \eta, q_f \in F or \eta = \epsilon\}$$

2.6.4 Parsing and Code Generation

A thorough investigation on what has already been done in hardcoding algorithms to implement FAs yields to two particular fields of study that use automata as basic

model for solving more specific problems: parsing and code generation. Both domains are similar even though the second tends to contain the first.

Code generation is precisely about writing programs that write programs[Herr03]. For the case of our work, a code generator is a program that generates a parser. The program is supplied with a specification of the input process (context free grammar) as well as a low level input routine (the lexical analyzer) to handle basic items (called tokens) from input stream [GRJA91]. The tokens supplied are organized according to the input structure rules (grammar rules). The parser generated is then useful to handle user input based on the rules that have been specified at generation time.

Parsing is the process of structuring a linear representation in accordance with a given grammar[GRJA91]. In general, a parser is an algorithm that determines whether a given input string is in a language, and produces a parse tree for the input if it is or an error message otherwise. Parsers are therefore syntactical analyzers for compilers. They are generated from context free grammars that characterize the language being used for testing the input string. There are several types of parsers depending to the properties of the language upon which the algorithm is based. One of the most widely used that require a nearly linear time are LR parsers. They are based on the left-to-right[3] technique, and they perform identification of he right most production. As described in [GRJA91], LR parsers are deterministic bottom-up[4] parsers having the following general characteristics:

- The determinism of such parsers lies in the fact that a shift involves no move.
- The right-most production expands the right-most non-terminal in a sentential form, by replacing it by one of its right-hand side. A sentence is produced by repeated right-most production until no non-terminal remains.

[3]The input is scanned from left to right

[4]Checking whether a string is part of the grammar is done from bottom to up as opposed to the conventional top-down approach

- Each step of a bottom-up parser working on a sentential form, identifies the latest right-most production and undoes it by reducing a segment of the input to the non-terminal it derived from. The identified segment is called *the handle*. Since the parser starts with the final sentential form of the production process (the input), it finds its first reduction rather to the left end.

Constructing the control mechanism of a parser can be implemented directly or using a code generator. The parser in most case is *table-driven*, consisting of drivers (routines) and one or more tables that describe its grammar as well as the lexical analyzer it is based upon. For table-driven parsers, the tables describing the grammar on which the parser is called *parse table*. In the parse table, the columns might contain terminal and/or non-terminal symbols, the rows generally contain the states of the push-down machine described by the grammar, and the entries contain the production rules. Parse table might considerably be large and complex, leading to important processing speed flaws at run-time. It is therefore of importance to explore alternative ways of representing the table to ensure efficient processing speed of the parser during syntactical analysis.

Most of the work that has been done for *hardcoding* parser algorithms relied on LR parsing as basic technique. The overall goal was to optimize the parse table processing speed by avoiding the implementation of table-driven parsers. In this case, the drivers of the parser are implemented is such a way that the parse table is imbedded into it using hardcoding implementation technique. It should be emphasized that such a strategy do not take into consideration implementation issues related to the lexical analyzer, since for various compilers lexical analysis is not part of parsing. However, since the overall data structure required to implement a parse table is fully determined by the structure of a Push Down Automaton (PDA), thus involving transitions functions implementation into hardcode, it might be useful to explore empirically the implementation of a lexical analyzer for example using hardcoding

technique.

There exist various parsing techniques and details can be found in standard introductory texts that cover the subject in more detail as in [AhSU86]. In the next subsections we summarize some of the parsing techniques that can be hardcoded.

Top-down parsing

Given an input string, the top-down parsing starts with the starting state and attempts to reproduce the input. One of the popular implementation technique use for top-down parsing is *recursive-descent parsing*. With this technique, a set of recursive procedures are executed to process the input. A procedure is associated with each nonterminal of a grammar. This mechanism therefore allude the fact that a recursive-descent parsers are hardcoded into procedures. Each procedure associated to a nonterminal decides which production rule to be use by looking at the lookahead symbol. The procedure also uses a production by mimicking its right side. More details on top-down parsing and its various versions can be found in [GRJA91], and [AhSU86].

Bottom-up parsing

This method is in general known as shift-reduce parsing. It attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the starting symbol of the grammar's rule (the top). The process therefore reduces the input string to the start symbol. Explicit details on Bottom-up parsing maybe found in [GRJA91, AhSU86]. Furthermore, we present LR parsing in more details since it appears to be necessary for an understanding of the next Section on *related work*.

LR parsing is an efficient, bottom-up syntax analysis technique that can be used by a large class of *CFGs*. The letter "L" denotes left-to-right scanning of the input,

and "R" stands for constructing the rightmost derivation in reverse. An LR parser is made of an input buffer, a stack, the LR parsing program (driver), and the parsing table which is itself made of a function *action* and a function *goto*.

The driver reads characters from the input buffer one at a time. It uses the stack to store a string of the form $s_0X_1s_1...X_ms_m$, where s_m in on top of the stack. Each X_i is a grammar symbol and each s_i is a symbol called a state. Each state summarizes information contained in the stack below it. A combination of the state symbol on top of the stack and the current input symbol are used to access the parsing table and perform the shift-reduce parsing decision. The driver determines s_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults $action[s_m, a_i]$, the parsing action table entry for the state s_m, and input a_i. The result can be one of the following four values:

1. shift s, where s is a state,
2. reduce by a grammar production,
3. accept, or
4. error.

The overall LR parsing algorithm can therefore be summarized as follows:

Algorithm 2. *LR parsing*

begin algorithm
pointer:= text[0];
while *(True)* **do**
s:= top(stack);
a:= pointer;
if *(action[s,a] = shift s')* **then**

push a;

push s';

pointer := pointer+1;

else if *(action[s,a] = reduce* $A \longrightarrow \beta$*)***then**

pop $2 * |\beta|$ *symbols off the stack;*

s':= top(stack);

push(A);

push(goto[s', A]);

output $A \longrightarrow \beta$ *;*

else if *(action[s,a]=accept)***then**

exit;

else

error();

exit;

end while

end algorithm

The algorithm above takes as input a string and a parsing table with functions *action* and *goto* for a grammar. It outputs a bottom-up parse tree if the input is part of the language described by the grammar; otherwise it outputs an error signal. This algorithm is conventionally said to execute table-interpretive LR parsing.

When a parser generator is provided with a BNF formula describing a *CFG* (an LR grammar), then it will automatically generate the parsing table, its stack, and the driver program for executing table-interpretive LR parsing. YACC is one of the best known LR parser generator that works precisely as described above.

The algorithm is implemented using a "driver" routine containing a loop. In each iteration of the loop, a single parse action is made: a terminal shift, a reduction, an acceptance of the input, or the emission of an error message. To determine the

action, the state on the top of the parse stack and the current input symbol have to be considered. It follows that three to four array accesses are usually sufficient to determine the next action. However, when more than one reduce action is needed, a list searching is required to determine the next action. The number of instructions required by the algorithm for a parsing action appears to be quite high. One way of seeking greater efficiency in implementing the LR parsing algorithm is by hardcoding aspects of the parsing table as well as the stack into the algorithm itself. Various authors have explored this broad strategy in various ways. The section below presents work that has been directed at using hardcoding to improve efficiency of the LR parsing algorithms.

2.7 Related Work

The subsections below present in a chronological order, the work of authors who, to the best of the author's knowledge, have been the most prominent in investigating the hardcoding of LR parsing algorithms.

2.7.1 Pennello

In 1986, Thomas J. Pennelo created a system that produced hardcoded parsers in assembly language [Penn86]. The system was a parser generator that takes as input a *BNF* description of an LR grammar and automatically generates a hardcoded parser in assembly language. The generated parser in turn takes as input a string and produces its parse tree if the string is syntactically correct or an error message otherwise. Pennello observed that by encoding the parser directly into assembly code the parsing time can be significantly improved.

Various strategies are used to implement the stack and the parsing table. While full details are available in the original article, two features of Pennelo's approach are

highlighted below.

Firstly, in the hardcoded assembly language, it is unnecessary to encode the stack as a separate data structure. The essence of the LR parsing algorithm given above, is that the stack contents (and specifically, the top of the stack) determines the next action to take in terms of the next input symbol. In the hardcoded counterpart to this algorithm, a labelled section of code is associated with each stack state. By directing control to appropriate labels, code that handles a given state is executed. For example, a *shift* action is carried out by executing a simple jump to a label associated with code for the destination state. When a reduction by a production p is indicated, a jump is made to the simple routine that handles p since such a reduction may occur in several states. A jump to another label is then executed to carry out instructions associated with the destination state as determined by the left part of the production. The source-code below provides possible forms for the assembly language code associated with specific states.

```
$Q_n$:  call Scan;   only if accessed by a terminal (get the next
input symbol).
     push address of label  $NTXQ_n$ on machine stack
        linear search for a terminal transition
        or
        mov Base_reg, Base
        call Find_T_transition
        jump to default reduction(rare)
        or
        binary search for default reduction(rare)
        or
        jump to error handler if no reductions possible
$NTXQ_n$: binary search for default reduction
```

```
        or
        in-line code to extract transition from NT_next

reduce state for production
p: $Q_p$: call Scan  ; only if
accessed by a terminal.
    mov R_NT,Left_part
    jmp $Reduce_{L-1}$
    or
    jmp Reduce_minus_1L
    or
    mov R_RPL, L-1
    jmp ReduceN
    or
    mov R_RPL,L
    jmp Reduce_minus_1N
$NSQ_p$: mov R_NT, Left_part
    jmp $Reduce_L$
    or
    mov R_RPL, L
    jmp ReduceN
```

Secondly, in Penello's system, the hardcoded implementation of the parsing table relies on a thorough analysis that is undertaken by the program generator. In effect, it analyzes the transitions specified by the parse table in any given state s to determine the resulting value of $action[s, a]$ for each possible terminal symbol a. The resulting value is either an injunction to shift to a specified new state, or to reduce to by a specified production, p. The code below shows an example of the code associated

with a state labelled Q23 indicating the "shift" transitions to be made if one of the nine terminal symbols 'a',...,'i', is encountered in that state. In this case, a linear search for the current symbol is used to arrive at the appropriate jump to be made when that symbol occurs in that state. If the current symbol is not one of the nine terminal symbols, then a jump is indicated to code that handles a "reduce" operation in regard to production number 20.

```
Q23: call Scan
;since accessed by a terminal.
     push address of NTXQ23
     cmp R_T,4 ; If 'a'
     je  Q11   ; shift to 11 on 'a'
     cmp R_T,5 ; If 'b'
     je Q12    ; shift to 12 on 'b'
     ...            ; assume similar code for 'c' to 'h'
     je Q19    ; shift to 19 on 'i'
     jmp NSQ20 ; Reduce by production 20
```

Similarly, the part of the parsing table represented by $goto[s, A]$ in the previously specified LR algorithm needs to be hardcoded to indicate, for a given state s, what the resulting state will be when reducing a given production A. The code below shows an example of the code associated with a state labelled NTXQ23, indicating the "reduce" transitions to be made if one of the ten nonterminals $A...I$ is encountered in that state. Note that in this case, a binary search for the relevant non-terminal symbol is used to arrive at the appropriate jump to be made. In Penello's system, the decision to use a linear search, a binary search or a jump table is part of the analysis to be carried out by the program generator.

```
NTXQ23: Binary  search for nonterminal transition
```

```
        cmp R_NT,19    ;If E
        ja L1
        cmp R_NT,16    ;If B
        jl Q33          ;shift to 33 on A
        je Q25         ;shift to 25 on B
        cmp R_NT,18    ;If D
        jl Q26         ;shift to 26 on C
        je Q27         ;shift to 27 on D
        jmp Q28        ; shift to 28 on E
L1:     cmp R_NT,21    ;If G
        jl Q29         ;shift to 29 on F
        je Q30         ;shift to 30 on G
        cmp R_NT,23    ;if I
        jl Q31         ;shift to 31 on H
        je Q32         ;shift to 32 on I
        jmp Q24        ;shift to 24 on K
```

The resulting hardcoded parser is reported to have shown a 6 to 10 factor improvement in speed over his table-driven system. However, this improved speed is achieved at a cost of a factor of 2 to 4 increase in space requirements. Full details on Pennello's work may be found in [Penn86].

2.7.2 Horspool and Whitney

Horspool and Whitney in [HoWh88] proposed a number of additional optimizations for hardcoding LR parsers. They used Pennello's attempt at optimizing decision sequences via appropriately implementing either a linear search, a binary search or the using of a jump table. But, in addition, they use an adaptive optimization strategy.

The strategy involves an analysis of the finite state machine (the goto function in the parsing table), as well as some low level code optimizations that increase the processing speed and decrease the code size.

They start by observing that many stack accesses (push, and pop) during LR parsing are redundant. Therefore using "Minimal Push" optimization techniques, they eliminate these redundant actions. The main observation is that only states with non-terminal transitions need to be pushed onto the stack. Such states are the only ones which may be consulted after a reduce production. This optimization implies an optimized way of dealing with the various right-recursive rules of the LR grammar. For example, a list containing n identifiers would require only n stack pushes, because pushes relating to the comma separator of list elements are suppressed in terms of their stack optimization strategy.

Another optimization used is the technique of *Unit Rule Elimination*. They note that grammars often contain productions in the form $A \rightarrow X$, where X represents either a terminal or nonterminal symbol. Such a production is called *unit rule*. LR parsers generated from grammars containing *unit rules* have an inflated number of states. The overall observation is that if there is no semantic action associated with a unit rule in the parsing table, the transition associated with such a unit rule may be deleted.

Having applied the above strategies during the construction of the LR parser, all that remains is to implement the parser directly into assembly code as suggested by Pennello.

Their parsers generator also creates a directly executable LR parser that operates at faster speed than the conventional equivalent, while simultaneously requiring a decreased amount of storage relative to the parsers generated by Pennello. The parsers created were 5 to 8 times faster than the equivalent table-driven parsers generated by YACC.

2.7.3 Bhamidiapaty and Proebsting

Bhamidipaty and Proebsting in [BhPr95] developed a YACC-compatible[5] parser generator that creates parsers that are 2.5 to 6.5 times faster that those generated by YACC. The tool creates directly executable hardcoded parsers in ANSI C (as opposed to Pennelo's assembly code), whereas yacc produces table-driven parsers. Their system creates code that is responsible for simulating the action of each state, this action being based upon the current input token. States labelled with tokens are called *shift states* and they require extra code to advance the lexical analyzer[6]. An extract of the hardcoded version of the implementation of a given state N is shown below:

```
State_N:
    stack  --- > state = N;
    If N is ashift state , then
       stack -> semantic = yyval;  // Put lexical semantic entry on stack
       token = yylex();              // Advance lexical analysis
       yyerrorstatus++;             //update error-recovery counter
    if (++stack == EOS)    gotostack_overflow;
state_action_N:                    //Error-recovery entry point
    switch(token){
    case Q:     goto state_X ;     // iff shift =  action[N,Q], X = goto[N,Q]
    case R:     goto reduce_Y;     // iff reducey = action[N,R]
    case S:     goto  error_handler;  // iff error = action[N,S]
    case T:     goto  YYACCEPT;       // iff accept = action[N,T]
    .
    .
```

[5]The generator creates LR directly executable LR parsers exactly as YACC, but the LR parser is not implemented using table-driven approach as YACC does

[6]Recall that the parser generator requires a reference to the lexical analyzer. The latter is used, in turn, to identify tokens for the parser.

```
.
// The action table determines the defaults action for N:
default:   goto error_handler;
or
default:   goto reduce_Z;
}
```

In the same way as Pennello's system, one piece of code (hardcode) is implemented for each production, avoiding therefore the use of external data from memory (the table). Nonterminal transitions are implemented using switch statements, enabling a jump to the appropriate state from the current state. An empirical evaluation of the method showed speedup factors ranging between 2.5 and 6.5. Not surprisingly, there was a code size increase of up to 128% but in practice, that represented less than 75KB in space.

2.8 Summary of the chapter

In this chapter, we have presented various aspects of computing that use an *FA* as a basic tool to solve a certain computational problem. We have shown the relationship of *FAs* with regular expressions, pattern matchers, and context free grammars. We have also shown that considerable work has already been done in hardcoding LR parsers by means of implementing efficient code generator programs. These take the *BNF* form of the LR grammar and generate a directly executable LR parser. Yet another variant of LR parsing which is implemented by hardcoding each state of the LR automaton as a procedure. In this case, a shift calls the procedure. The manipulation of the stack is avoided using embedded reduced action into the procedure. Details about *recursive-ascent parsing* may be found in [AhSU86] and [Krus88].

The present work does not extend what has already been done in hardcoding LR

parsers. Rather, we restrict ourselves to hardcoding of finite automata in general. Thus, our work relates to hardcoded possibilities for lexical analyzers, regular expression recognizers, pattern matchers, and any other string recognition problem where an *FA* is the underlying model.

The main concern in this work is the *processing speed of FAs*. By this we meant the time taken by a device to accept or reject a string. We focus on the complexity metric defined above, trying to minimize its value as much as possible using appropriate programming techniques. The next chapters deal with the theoretical and practical aspects of the implementation and experimentation of hardcoded algorithms on finite automata. Of course, experiments are conducted using as basis the traditional table-driven implementation approach that is used by most automata implementers.

Chapter 3

Problem Domain Restriction

3.1 Introduction

In this chapter, we present abstractions to characterize the problem of concern. The chapter starts with a general specification of both the table-driven and the hardcoded algorithms to implement an *FA*. A theoretical evaluation is made on both methods. It is shown that at the theoretical level no order of magnitude differences between hardcoding and the table-driven approach are observable. However, such an analysis does not illuminate likely behaviour in practical situations. To do this, an empirical analysis of the behaviour of the two approaches is required. Section 3.5 justifies the restriction of the problem domain in a particular way in order to facilitate such an empirical analysis. The actual empirical analysis is discussed in a later chapter. The chapter ends with a specification of how each of the two algorithms would be specialized when applied to the restricted domain.

3.2 The Table-Driven Algorithm

A table-driven algorithm is the usual basis for ascertaining whether a string *str* is a member of the language generated by an *FA*, M. Consider a string, *str*, of length $len > 0$, and a table $transition[i][j]$ that represents the transition function of M, where $0 \leq i < numberOfStates$ and $0 \leq j < alphabetSize$ [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 3 shows how the string *str* is conventionally tested for membership of the language accepted by M.

Algorithm 3. *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
```

[1] 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.

In terms of complexity, Algorithm 3 largely depends on the length of the string being tested for recognition. Thus the worst case scenario requires $O(len)$ time to be completed. Our aim is to improve the algorithm using hardcoding. The section below depicts the hardcoded algorithm of a given FA that serves as a basis for our investigation.

3.3 The Hardcoded Algorithm

Hardcoding the recognition of a string avoids the use of a table as a parameter, but generates instead, code that is specifically characteristic of a given transition table. The idea is to break the transition table into instructions, so that the resulting algorithm does not use external data, and has integrated its required data. To do so, we analyze each state of the automaton by grouping into a set all symbols of the automaton's alphabet that may trigger a transition in that state. We call $validSymbols_i$ the set representing all symbols that may trigger a transition in state i. We also call $nextStates_i$ [2] the set of all those states that can be reached if an element of $validSymbols_i$ triggers a transition in state i. We incorporate labels into the algorithm that mark the piece of code dealing with each state such that, at a given state k, if the current input symbol belongs to the set $validSymbols_k$, then a transition is made to its corresponding next state in the set $nextStates_k$ or $true$ is returned if the end of str has been reached; otherwise $false$ is returned.

Consider a string, str, of length $len > 0$, and an FA, M of $numberOfStates \geq len$ states. Algorithm 4 depicts how the string str is tested for membership of the language accepted by M. Note that the a statement of the form ***goto*** $nextStates_i$***;*** is intended as shorthand for a number of successive conditional instructions, each indicating

[2] The fact that we are dealing with deterministic finite automata requires that a unique element of $validSymbols_i$ corresponds a unique element of $nextStates_i$ to be transited to. This is a form of surjective relation between $nextStates_i$ and $validSymbols_i$.

jumps to specific labels, depending on the precise value of $str[i]$.

Algorithm 4. *Hardcoding string recognition*

```
function recognize(str):boolean
  state_0:
    if str[0] ∉ validSymbol_0
      return(false);
    else if len = 1
      return(true);
    else
      goto nextStates_1;
    end if
  state_1:
    if str[1] ∉ validSymbol_1
      return(false);
    else if len = 2
      return(true);
    else
      goto nextStates_2;
    end if
  state_2:
    if str[2] ∉ validSymbol_2
      return(false);
    else if len = 3
      return(true);
    else
      goto nextStates_3;
    end if
```

```
...
...
...
state_{numberOfStates-1}:
  if str[numberOfStates-1] ∉ validSymbol_{numberOfStates-1}
    return(false);
  else
    return(true);
  end if
end function
```

The production of algorithm 4 may seem cumbersome. However, it might offer some speed advantages that we will investigate in the next section. The algorithm's execution time clearly depends on the length of the string being tested for recognition. The worst case complexity is $O(len)$. The next section provides a theoretical comparison between algorithm 3 and 4.

3.4 Comparison of Hardcoding and Table-Driven Algorithms

Three factors may be taken into consideration when comparing the two algorithms:

- the code size;
- the memory load as represented by the data structures required for the algorithms; and
- the complexity as measured by an order of magnitude estimate of the number of instructions that have to be executed for a given input.

	Hardcode	Softcode	Remarks
Instructions required	$4 \times numberOfStates$	$5 + 2 \times len$	Softcode seems better
Data required	1	$2+alphabet \times States$	Hardcode seems better
Complexity	$O(len)$	$O(len)$	Both equivalents

Table 3.1: Evaluation of algorithm 3 and 4

Any other factor one can specify falls under one of the above mentioned. The number of instructions determines the code size whereas the amount of data used in the algorithm determines the memory load. In the extreme, if the code size becomes too large, it may not be possible to compile it. On the other hand, if the memory load is excessively large, it might not be possible to execute the compiled program. In between scenarios might also occur, where various levels of cache memory are used. However, for the present, these extreme scenarios will be ignored. In later sections, the impact of cache memory will be investigated.

It is possible to perform a comparative evaluation of algorithm 3 and 4, based on the above, as depicted in table 3.1.

The two algorithms share the same order of magnitude complexity, which – for both algorithms – is linear in the number of characters in the string to be recognized by the *FA*. The code size and the memory load are also important factors here, and there are clear differences between the two approaches. The table entries in regard to these factors are to be justified below.

Hardcoding requires very little additional data, due to the fact that much of the data has directly been hardcoded. This is justified by the presence of the value 1 in the corresponding cell in the table. In effect, as shown in algorithm 4, at each state, there is a single conditional statement that performs a test on the current symbol of the string ($str[i]$). The symbol is loaded in the memory; this justifies the fact that 1 data (one memory access) is performed at each state. Any other operations are considered to be hardcoded. The *len* of the string is automatically evaluated at

run/compile time, $validSymbols_i$ is known in advanced, therefore, hardcoded.

The table-driven approach, on the other hand, heavily depends on data, thus memory load. The algorithm requires just before starting the loop, 2 data access in the memory (from *state* and $stringPos$); the algorithm also requires entries in a table representing the transition between one symbol in the alphabet and an a given state, thus the factor $alphabet \times state$ where *alphabet* represents the number of elements in the alphabet of the grammar that describes the *FA*. This justifies the fact that $2 + alphabet \times state$ data are required in the worst case by algorithm 3.

Very few instructions are needed to implement table-driven approach, but that is not the case for hardcoding. The latter requires approximately $4 \times numberOfStates$ instructions, which is very consistent compared to the table-driven which only requires $2 \times len$ ($len \leq numberOfStates$). Theoretically, if memory load constitutes our assessment standard, hardcoding implementation is far better than the traditional table-driven implementation of finite state automata. The inverse is verified when the criterion is the code size. With better compression [3] techniques applied at the encoding stage, it is also possible to overcome the problem of size as mentioned in [HoWh88]. This results in a very efficient hardcoding algorithm, compared to the traditional table driven implementation.

Having established theoretically the advantage of using hardcode over table-driven method, the next section performs some observations on table 1 that results in the restriction of the problem domain.

3.5 Problem Restriction

Observations on table 1 suggest that measurements are approximately linear. The following remarks can be made:

[3] The compression of hardcode is beyond the scope of this work. At present, the major concern is to establish the efficiency of hardcode in terms of processing speed and not code size.

- For hardcoding:
  - the number of instructions is approximately $4 \times numberOfStates$, meaning that there are four basic instructions [4] that are executed in each state of the automaton.
  - the amount of data required in the memory is simply the string to be recognized, all other data being directly hardcoded and stored as values in the algorithm's variables (len, $validSymbol$);
- For softcoding:
  - The number of instructions is approximately $2 \times len$ – i.e. two instructions are executed per symbol of the string to be recognized;
  - The amount of data required is $alphabetSize \times numberOfStates$; so that in any given state, each symbol of the string has its allied entry in the table.

Based on the above, the following restrictions suggest themselves.

- Rather than studying an entire string, it seems reasonable to analyze the response to a single symbol of the alphabet at a time and to draw tentative conclusions about the behaviour of an entire string on this basis. At a later stage, a more complete investigation into the behaviour of the algorithm in relation to a complete string can be explored.
- Rather than analyzing a complete automaton that may consist of a large number of states, it seems reasonable to restrict attention to a limited portion of the automaton – in fact, to a single arbitrary state – without the loss of generality.

[4] This is, of course, an abstraction since in each case, the instruction, goto $nextStates_1$ in fact implies the execution of several additional instructions, as has previously been noted.

Algorithm 4 consists of near-identical chunks of code, each chunk dealing with a state of the automaton. In terms of processing speed, each chunk will consume the same time. Therefore, analyzing the performance of an arbitrary chunk of the algorithm appears to be a reasonable strategy, since the entire processing speed of the algorithm will simply be a multiple of the time taken for a single chunk.

From the point of view of the present discussion, the most important portion of algorithm 3 is the iteration to be performed when processing a string. That portion of the algorithm corresponds in fact to the process of recognition of a single symbol, repeated several time according to the length of the string being recognized. It is thus possible restrict further analysis to a single iteration of the loop, which in fact corresponds to the analysis of some single state of the automaton for a given symbol of the string.

Without the loss of generality, therefore, the original problem can be restricted to the study of some arbitrary single state of some finite state automaton. Conclusions will be drawn later about complete automata that may have several states.

3.6 Single Symbol Recognition

In algorithm 3, 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. A consideration of the time taken to process a single symbol instead of an entire string is 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 is considered to have no transition at all on the remaining alphabet symbols. Figure 3.1 depicts one such a state, where the five out-arcs represent five legal transitions associated with different input alphabet symbols.

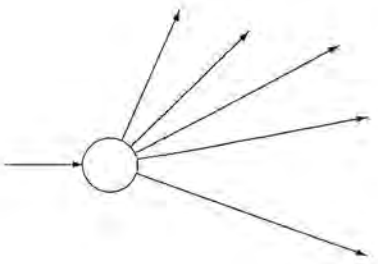

Figure 3.1: A state in the transition diagram of some finite automaton

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

Figure 3.2: A transition array for a state of some automaton

Figure 3.2 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 *FA*. The array is indexed by the *FA*'s alphabet: $\{a, b, c, d, e, f, g, h\}$[5]. 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). Figure 3.2 therefore indicates a state where there will be a transition to state 8 upon encountering an a symbol in the input, no transition is possible for b, c

[5]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.

or d inputs, a transition to state 10 upon input e, etc. Algorithm 5 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 5. *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 3 that has been trimmed down to deal with one symbol in some arbitrary state described by the transition array.

3.7 Hardcoding Single Symbol Recognition

Hardcoding generates code that is specifically characteristic of a given transition array. Algorithm 6 shows an example, in pseudo-code, of the hardcode to deal with the single state transition array depicted in figure 3.2. Note that this hardcode has to change whenever a transition table with different entries is to be used.

Algorithm 6. *Hardcode of a transition array of finite automaton in the figure 3.2*

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 5 and 6 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 5 and 6 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 5 and five of algorithm 6). These are described in chapter 5.
- The algorithms' performance may vary from one hardware platform to another. Hardware considerations are discussed in the next chapter.
- The algorithms' performance should be compared in relation to many different input scenarios. Algorithm 7 in section 5.2 describes how various randomized transition arrays were generated for the study.

3.8 Summary

In this chapter, we have presented table-driven and hardcoding specifications of finite state automata and theoretical comparisons have been performed. A rationale has been given for restricting the problem domain to a single state of some finite state automata for some arbitrary symbol. We now need to present tools required to conduct practical experiments on algorithms 5 and 6. This constitutes the theme of the next chapter.

Chapter 4

Tools and Methodology

4.1 Introduction

The realization of a practical comparative study requires the selection of basic tools needed to achieve the goals; it also requires the setup of means by which experiments will be carried out. This chapter aims to address those issues. Tools required for cross-comparing hardcoding and softcoding of finite state automata are presented, and a strategy is designed for the experiments.

4.2 Hardware Considerations

Finite automata implementation can be carried out on any personal computer as well as special purpose computing devices designed to handle specific state transition tasks. A personal computer was regarded as a good choice for the experimentation, not only because of the convenience of being readily available, but also because programs can be written and tested with little effort and at relatively low coast. The configuration of the computer system was not considered to be of prime concern because the overall objective of the study relates to the processing speed and not to the strength of

the computer's other components. Several kinds of processors were available for our experiments. Those we had to choose from included AMD, Celeron and Intel processors. It is known that AMD and Celeron are both Intel compatible because they have integrated some emulators that handle Intel-specific instructions. In the absence of any selection criteria that strongly commended the use of one processor over another, it was decided that the use of Intel processors would serve as an adequate and convenient tool to be used for the empirical study. The aim has been to produce a quantitative evaluation of two *FAs* implementation techniques, independently of the type of hardware being used. Consequently, it is considered unlikely that the results the experiments conducted would be significantly different if any other compatible processor had been used.

Our experiment was performed on the Intel architecture (IA 64)– specifically a Pentium 4 at 1 GHz with 512 MB RAM and 10GB of Hard drive. While the broad conclusions of the study are unlikely to be affected if other hardware had been used, verification of this claim remains an aspect for further study.

4.3 Software Considerations

Any operating system platform could have been used to conduct the experiments. We have chosen to work under Linux because its RedHat version incorporates free software which can be use to encode programs. In addition, any programming language could have been used to implement finite state machines and conduct proper experiments. As high-level language, we considered C++ to be suitable because it offers many optimization options. The g++ compiler of the free software foundation group gnu was used. Netwide Assembler (NASM) was our low-level assembler of choice, merely because of its simplicity. Any other assembler could have been chosen, based primarily convenience of use. Having established the suitable programming

environment and operating system platform, the software needed to produce the results of the experiment in graphical form was selected. We had to make a choice between gnuplot and Microsoft Excel for Windows because they both offer plotting capabilities on data collected. For simplicity reasons, we chose Microsoft Excel.

4.4 The Intel Pentium Read Time Stamp Counter Instruction

Since the experiment involved cross-comparison of times taken for different algorithms, it required the use of a software instruction to measure times. On Intel microprocessors, the so-called time stamp counter keeps an accurate count of 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]. Whether invoked directly as a low-level assembly instructions, or via some high-level language instruction, the RDTSC[1] instruction allows one to read the time stamp counter before and after executing critical sections of code, and thereby to determine approximately the time taken to execute these critical code sections. It was extensively used in the coded experiments of this study.

4.5 Random Number Generation

In this experiment that was to be conducted on *FAs*, or more precisely on an arbitrary state of some *FA*, we needed a way to simulate the behavior of the *FA* in such a state. Lacking a repository that contained a representative sample of finite automata to conduct the experiments, we have chosen to randomly generate our own arbitrary

[1]Read Time Stamp Counter

state of a finite automaton. By a randomized state is meant one that is characterized by:

- A random set of symbols representing the alphabet from which is selected a random subset of symbols. Each of which triggers a transition to an randomly determined next state. Whereas the complement of the random set of symbols, represents symbols that do not trigger transitions at all.

- The size of the random subset of symbols that could trigger transitions, was also determined randomly. This number determined the *density* of the transition function (i.e. the proportion of valid transitions).

To be able to obtain reasonably representative data, we needed to generate on this random basis a large number of such arbitrary single states and then to use them for experiments. The use of an optimal random number generator program was therefore of interest. According to [PTVF02], the efficiency of a random number generator algorithm largely depends upon the size of the data to generate as well as on the kind of data to be generated. Three main options for random number generation are proposed in [PTVF02]; Source-code 9, 10 and 11 in appendix A, depict each of them written in C++. The type of values we aimed to produce was in the order of hundreds. The series to be generated does not need to follow specific statistical laws other than to be random. We aimed to produce a series that does not contain repetitive values, so that all possible symbols were used in the experiment. We chose the function ran2 (source-code 10) for its efficiency, and for its ability to produce in a single run fairly randomized numbers. These properties are discussed in [PTVF02].

4.6 Methodology

We aimed to cross-compare various hardcoding implementations against one another and also against the table-driven method. The implementation of the table-driven specification is straightforward and any programming language may be used. Further details need to be provided for the hardcoding technique, because several implementations both in high-level and low-level languages were considered.

Using a high level language, algorithm 6 can be coded in two ways: either switch statements or nested conditional statements (if...then...else...) could be used. It was decided to investigate three strategies in a low-level language, namely: the use of jump tables, the implementation of a linear search and the use of so-called direct jumps. Figure 4.1 depicts the overall process flow used to compare the various hardcoded implementations and the table-driven implementation. Figure 4.1 gives an overview of the main processes involved in the design and execution of the experiment. The process started by 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 5). It is also used to hardcode (i.e. implement) algorithm 6 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 4.1 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 *FA* 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 5 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 4.1 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.

4.7 Chapter Summary

We have presented in this chapter practical tools needed to conduct the intended experiments. Issues relating to hardware, software, and programming language selection have been presented. The overall methodology indicating the techniques to be used for cross-comparison was presented.

The next chapter is devoted to the implementation details related to each algorithm specified in the previous chapter as well as to the implementation of the methodology of the previous section using the tools depicted in this chapter.

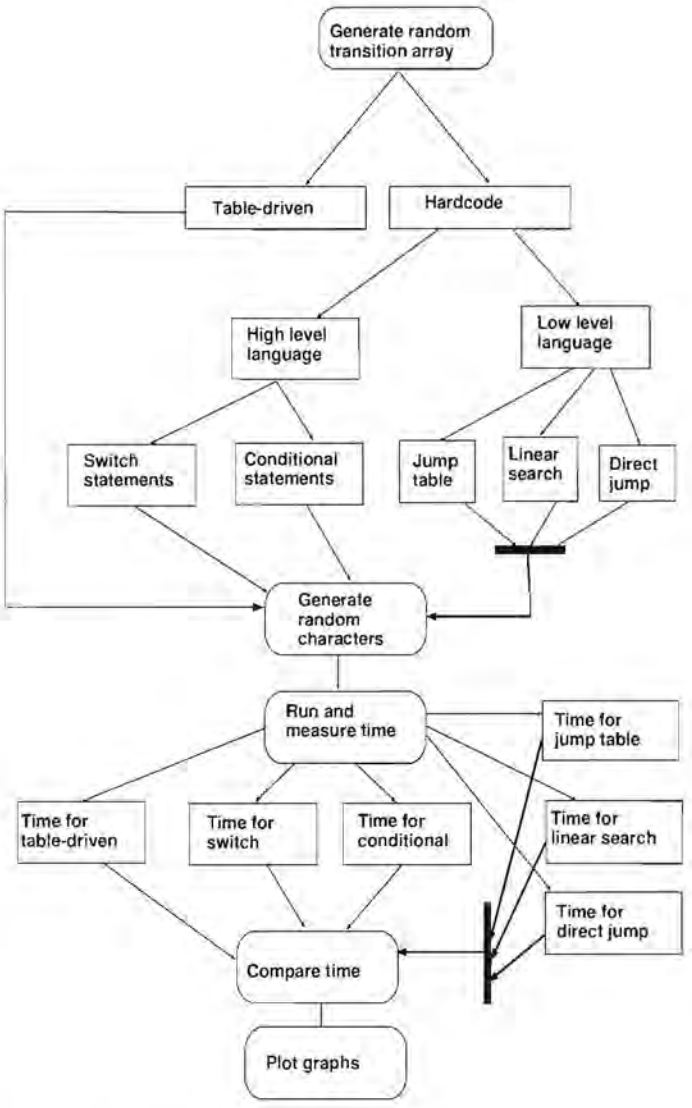

Figure 4.1: Process diagram indicating how the hardcoded implementation were compared to the table-driven implementation

Chapter 5

Implementation

5.1 Introduction

This chapter is about the overall implementation of our work. It focuses on the actual implementation of each critical session of the flow diagram presented in Figure 4.1 of the previous chapter. The design of the random transition array is presented as well as details about how we randomly generate symbols of the alphabet to be tested for recognition. The chapter ends with the presentation of each of the six coded implementations that constitute the experiment's building blocks. Thus, the table-driven encoding in a high-level language is depicted. Various hardcoding experiments in high level language and low level language then follow. The chapter ends by indicating how we handled the actual experiment having each strategy defined. The approach to data collection for cross-comparison purposes is specifically raised.

5.2 The Random Transition Array

For the purposes of the experiment, it was necessary to generate randomized transition arrays. Such an array is based on the following random selections:

- the alphabet size of the *FA*, say *alphabetSize*; and
- the maximum number of transitions, $numberOfTransitions \leqslant alphabetSize$, that can be made from the simulated single state.

For the present purposes, it was considered appropriate to limit the alphabet to integer values in the range $0 \leq alphabetSize \leq 255$[1]. 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 have been 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 less than *alphabetSize*. Such a random number is generated *numberOfTransitions* times. No provision is made to enforce the selection of a different random number in each generation, which means that sometimes an array index position may be reselected. Each array index position could be associated with 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 $alphabetSize - numberOfTransitions$ 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. Algorithm 7 shows how the randomized transition array is generated.

[1]Of course, these integer values could be regarded as bit strings within a byte, and could be given any interpretation required by a particular application, e.g. they could be viewed as Unicode characters, etc.

Algorithm 7. *Generation of random transition array*

function *genRandomArray()*:**transitionArray**

i := 0;

alphabetSize := ran2(0,255);

numberOfTransitions := ran2(0,alphabetSize);

{Initialization of the transition array}

for *i:=0* **to** *alphabetSize* **do**

transition[i] := -1;

end for

while*(i < numberOfTransitions)* **do**

index := ran2(0,alphabetSize);

transition[index] := ran2(0,maxInt);

i := i+1;

end while

return(transition);

end function

5.3 Table-Driven Implementation

A C++ version of the table-driven implementation takes as input a transition array, as well as a single randomly generated input character. Source code 1 below illustrates the actual encoding of algorithm 5. Notice that to prevent the C++ optimizing compiler from bypassing the execution of the statement related to the random simulation of a valid transition or not, we introduced a function *fake_transition()* that contains some "fake" statements, forcing the compiler to actually consider the assignment that determines a next state. The reason behind all of this is that the g++ compiler is very efficient in generating optimized executable code from source code.

One of its strength is to bypass any statement it may judge not to be useful to the entire program being compiled. Therefore, a sequence of statements such as

```
begin = rdtsc();
nextState = ptrTransition[k];
end =  rdtsc();
tsc = end-begin;
```

will inevitably produce a $tsc = 0$. This because the compiler regards *nextState* to be of no use in the code, and thus bypasses the statement $nextState = ptrTransition[k]$.

It was therefore necessary to look for a means of "forcing" the execution of such a statement. This was achieved by introducing the function "*fake_transition()*". The function prevents the compiler from bypassing an important statement that constitutes the very statement being tested in our work.

We consider here the time to execute a "*fake_transition()*" call. For the present experiment, this represents the time taken to receive an assignment either of an arbitrary next state or of -1 (indicating that the next state does not exist)[2]. Acceptance or rejection is actually determined by the content of the variable *nextState* after executing "*fake_transition()*". In the case of our experiment, a transition was said to be invalid for a given input symbol if the corresponding value of the transition function at that current state with the given symbol was -1. Any other value was a positive integer that represented a valid transition to an arbitrary state of the automaton.

Source-code 1. *C++ extract of table-driven symbol recognition algorithm*

```
void fake_transition(char a, long &next){
long val;
```

[2]Of course, in normal circumstances, the *fake_transition()* invocation consumes some time that affects the overall time measured to accept or reject a symbol. This situation was taken into account in the later experiments as described in chapter 7.

```
val = ptrTransition[a];
next = val;
}
int main(int argc, char *argv[]){
  ...
  // assume alphabetSize below is known (global variable)
  /*Generate the random transition array using
   genRandomArray and store in ptrTransition */
   ptrTransition = genRandomArray();
   for (k = 0; k < alphabetSize, ++k){
        nil = rdtsc();
        nil = rdtsc()-nil;
        begin = rdtsc();                // read time stamp counter
        fake_transition(k,nextState);
        end = rdtsc();                  // read time stamp counter
        tsc = end-begin-nil               // compute the processing speed
    }
  ...
return 0;
}
```

5.4 Hardcoded Implementations

As shown in figure 4.1, two branches were taken in regard to the implementation of the hardcoding experiments. The first branch refers to two high-level language hardcoded implementations, and the second, to three low-level (i.e assembly) language hardcoded implementations. The figure thus alludes to a total of five separate programs that

generate hardcode. Each of these was written in C++. Each such program takes as input, some transition array indicating how transitions from a single state of an *FA* are to be made, and generates as output, the corresponding hardcode to handle a single input character in terms of such a transition array.

Source code 2 and 3 illustrate how these programs to generate hardcoded programs were written in C++. In the former case, the hardcode generated is in a high-level language while, in the latter case it is in a low-level language. Note that similar approaches were used for the other three cases.

Source-code 2. *C++ extract of high-level language hardcode generator based on nested conditional statement to recognize a single symbol*

```
int main(int argc, char *argv[]){
  ...
testFile = fopen(fileName, "w");
// headers files
fputs(''#include \'''', testFile);
fputs(''rdtsc.h'', testFile);
// and any other header file required
  ...
fputs(''int main(void){\n'',testFile);
fputs(.... variable declarations);
fputs(.... more variable declarations);
  ...
fputs(''begin = rdtsc();      // read time stamp counter \n'',testFile);
// auto  generate any single line of the nested conditional statement
for(auto int k = 0; k < alphabetSize; k++){
   // we use the randomly generated transition table to construct the
```

```
 // hardcoded program
if (ptrTransition[k] != -1){
   firstValidTrans++;
   if (firstValidTrans == 1){
       // begin the first line of the nested conditional statement
       fputs(''  if (a == '',testFile);
       fputs( tostr(k),testFile);
       fputs('') nextState = '', testFile);
       fputs(tostr(ptrTransition[k]),testFile);
       fputs(''; \n'', testFile);
   }
   else
   {
       fputs('' else  if (a == '',testFile);
       fputs( tostr(k),testFile);
       fputs('') nextState = '', testFile);
       fputs(tostr(ptrTransition[k]),testFile);
       fputs(''; \n'', testFile);
   }
  }
}
  // default statement
       fputs('' else '',testFile);
       fputs('' nextState = -1; \n'', testFile);
  //
  fputs(''end = rdtsc();          // read time stamp counter \n'',testFile);
  fputs(''tsc = end-begin;      // read time stamp counter \n'',testFile);
```

```
  ...
  fputs(''return 0; \n'');
  ...
  fclose(testFile);
}
```

Source-code 3. *C++ extract of low-level language hardcode generator based on jump table to recognize a single symbol*

```
int main(int argc, char *argv[]){
  ...
testFile = fopen(fileName, "w");
// headers files
fputs(''\%include \''asm_io.inc \'' \n'', testFile);
fputs(''\n'',testFile);

fputs(''segment      .data\n'',testFile);
fputs('';initializations'',testFile);
  ...
fputs(''TABLE   dd    '',testFile);
if (ptrTransition[0] == -1)
    fputs(''case_default\n'',testFile);
else
    for (auto int k = 1; k < alphabetSize; k++){
      if (ptrTransition[k] != -1){
          fputs(''       dd       '',testFile);
          fputs(strcat(''case_'',tostr(k)),testFile);
      }
```

```
        else
            fputs(''        dd    case_default\n'', testFile);

    }
...
fputs(''segment          .bss\n'',testFile);
...
fputs(''segment          .text\n'',testFile);
fputs(''                 global asm_main\n'',testFile);
fputs(''asm_main:        \n'',testFile);
  ...
fputs(''rdtsc\n'',testFile);
fputs(''mov        [ebp+8],eax \n'',testFile);
fputs(''mov        [ebp-8],edx \n'',testFile);
fputs(''mov        eax,[epb-4] \n'',testFile);
fputs(''shl        eax,2 \n'',testFile);
fputs(''mov        esi,TABLE \n'',testFile);
fputs(''add        esi,eax \n'',testFile);
fputs(''jmp        [esi] \n'',testFile);
...
...
fputs(''case_default:\n'',testFile);
fputs(''              mov     edx,-1\n'');
fputs(''              jmp     next\n'');
for(auto int k = 0; k<alphabetSize; k++){
    if (ptrTransition[k] != -1){
       fputs(strcat(''case_'',tostr(k)),testFile);
```

```
        fputs(''':\n'',testFile);
        fputs(''        mov      edx,'',testFile);
        fputs(ptrTransition[k],testFile);
        fputs(''\n'',testFile);
        fputs(''        jmp      next\n'',testFile);
    }
}
...
  fputs(''rdtsc\n'',testFile);
  ...
  fclose(testFile);
}
```

The subsections below discuss each of the hardcoded implementations, and explain the reason for our choice.

5.4.1 Use of the Nested Conditional Statements

An implementation based on nested conditional statements appears to be a natural and easy way to encode a hardcoded algorithm for *FAs*. It is an almost direct translation of algorithm 6 into C++. As can be seen in that algorithm, a conditional statement is used to check whether a symbol is valid in the transition table or not. Source code 4 gives an extract of the actual implementation in C++ for some randomly generated transition array. In this particular array, character 0 causes a transition to state 8, characters 1, 2, 3 and 4 are not associated with a transition, character 5 causes a transition to state 10, etc. The program thus records the time stamp counter value in *tsc* for each possible input character value a that is less than *alphabetSize*.

It may be observed that the value of *tsc* will be the least when *a* corresponds to the first accepting symbol in the randomly generated transition array. As the symbol being evaluated is sought after more deeply in the nested conditional statement, so the time required to locate it increases. Therefore, the worst case scenarios arises when the symbol *a* is not an accepting symbol in the transition array. In that case, all branches of the conditional statement will be tested sequentially before reaching the default case. When the number of branches become relatively large then such an implementation seems likely to be less efficient on average in comparison with the switch statement implementation that is presented in the next section.

Source-code 4. *C++ extract for nested conditional statement implementation*

```
...
for (a =0; a<alphabetSize; ++a){
  begin = rdtsc();
    if (a == 0) nextState = 8;
    else if (a == 5) nextState = 10;
    ...
    else  nextState = -1;
   end = rdtsc();
   tsc = end-begin;       //record tsc
}
...
```

5.4.2 Use of the Switch Statements

The switch statement is a high-level alternative to implementing nested conditional statements. It would appear to be especially appropriate when characters are to be recognized in general. However, as will be seen when the experimental results are

presented, the use of nested conditional statements appears to be inefficient than switch statements in terms of processing speed, especially when dealing with the default case.

Source-code 5 shows an extract of the encoding of a hardcoded version of algorithm 6 based on a switch statement. The randomly generated transition array that is to be hardcoded is the same as that of the previous hardcoded example. In the source code, for each value of a between 0 and *alphabetSize*, a *tsc* values is recorded.

A switch statement being a branching structure, one may predict same execution time for each accepting symbol to be evaluated. Such a prediction is made without taking into consideration the way the structure is compiled down at a lower level.

Source-code 5. *C++ extract for switch statement implementation*

```
...
for (a = 0; a < alphabetSize; ++a){
 begin = rdtsc();
     switch(a)
    {
      case 0: nextState = 8; break;
      case 5: nextState = 10; break
      ...
      default: nextState = -1; break;
    }
 end = rdtsc();
 tsc = end-begin;        // record tsc
}
...
```

5.4.3 Use of a Jump Table

The use of a jump table in assembler was suggested by the results of disassembling 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 such an optimized jump table is shown in source-code 6.

A table is defined in the data segment. Each table entry contains the 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.

In the source code below, a loop is performed for each symbol stored at memory location *ebp-4*. That location is used as a control variable, even though evaluation to check whether the loop has reached the end is done using the register *eax*. At any step in the loop, *eax* is assigned the value of the current symbol. We then need to evaluate the corresponding block of code for the symbol being examined. This is done by firstly shifting the content of *eax* by 16 bits (2 bytes) to get a double word using *shl* (shift left). Next the resulting content of *eax*, added to the address of *TABLE*, is stored in *esi*. Such a series of instructions is justified by the fact that each entry of *TABLE* occupies a double word (*dd*), whereas the initial content of *eax* is a single word. We then need to shift it left by two bytes to make it have the same length as an entry in *TABLE*, then add that value to the start address of *TABLE* to obtain the exact entry point corresponding to code relating to the symbol being tested. This is done without any ambiguity because each symbol is numbered from 0 to say *alphabetSize-1*.

After obtaining the address of the block of code to which to jump, the program records the start value for *tsc*, then executes the corresponding block of code, and records the final value for *tsc* respectively. The duration of the processing speed of the current block is then calculated and recorded. The jump table is implemented in such a way that all default entries are part of the table, but only one default block of statements is labelled in the program. Thus, for example, the symbols 1, 2, 3, 4, and 6 (in the Source-code below) do not have a next state, in order words, they are rejecting symbols and should be treated in terms of the default code. While they therefore have different entries in the jump table, each table entry refers to an identical address, designated "*case_default*". There is only one action associated with any such rejecting symbol: the action carried out in terms of the unique block labelled by *case_default* in the Source-code.

Source-code 6. *Sample code for jump table implementation in NASM*

```
segment      .data
;define the TABLE entries
TABLE    dd  case_0
         dd  case_default   ;rejecting symbol
         dd  case_default   ;rejecting symbol
         dd  case_default   ;rejecting symbol
         dd  case_default   ;rejecting symbol
         dd  case _5
         dd  case_default   ;rejecting symbol
         ...
         ...
         mov dword [ebp-4], 0 ; first symbol of the alphabet
         mov dword [ebp+24], alphabetSize
```

```
        mov eax, [ebp-4]
begin_loop:
        cmp eax, [ebp+24]
        je  end_loop
        ;
        ; read start tsc value
        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          ; for any rejecting symbol
        jmp next
case_0:
        mov edx,8
        jmp next
case_5:
        mov edx,10
        jmp next
...
next:
        ;read final tsc value
        ;calculate duration
...
       mov  eax, [ebp-4]     ; restore the current symbol into eax
```

```
        inc  eax              ; increment to the next symbol
        jmp  begin_loop
end_loop:
    ...
```

5.4.4 Use of a Linear Search

Linear search involves a sequential comparison of the various cases. Its structure is fairly obvious and is suggested by the disassembly of the high-level conditional statements. The implementation is straightforward. It involves a sequential series of comparisons and jumps to the appropriate label, according to the current symbol. Source-code 7 below depicts an extract of NASM code that implements the linear search.

The program loops on the alphabet symbols and the time is recorded for each symbol recognition action. The code shows that comparison is sequential. The comparison is performed only on symbols that trigger a transition to some state. The default case handles the case related to rejecting symbols. It follows that a rejecting symbol is only noticed after performing all the preceding comparisons on all accepting symbols. However, a symbol that matches at the beginning of the comparison structure is processed at a minimal speed compared to any other symbols that may match somewhere in the middle or at the end of the control structure.

Source-code 7. *Sample code for linear search implementation in NASM*

```
        ...
        mov dword [ebp-4], 0 ; first symbol of the alphabet
        mov dword [ebp+24],  alphabetSize
        mov eax, [ebp-4]
    begin_loop:
```

```
        cmp eax, [ebp+24]
        je  end_loop
        ...
        ; read start tsc value
        ...
        mov eax, [ebp-4]     ; store the value of the symbol in eax
        cmp eax, 0
        je  near case_0
        cmp eax, 5
        je  near case_5
        ...
        je  near case_default
; specify the different case statements
case_0:
        mov edx,8
        jmp next
case_5:
    mov edx,8
        jmp next
...
case_default:
        mov edx, -1
        jmp next
next:
        ;read final tsc value
        ;calculate duration
```

```
        mov  eax, [ebp-4]     ; restore the current symbol into eax
        inc  eax              ; increment to the next symbol
        jmp  begin_loop
end_loop:
        ...
```

5.4.5 Use of a Direct Jump

This implementation strategy 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. Having noticed that each block can be represented in fixed size, the idea is to calculate the absolute address of a block using the symbols that it represents in the alphabet. For that reason, we used the formula $addr = ip + c + w * R$ where ip represents the current address of the instruction pointer, c a constant representing the size of any gap that might exist between ip and the block to which to jump, w is the size of block, and R is the register containing the current symbol. In the code below (Source-code 8), c is taken as 7 bytes and w is taken as 10 bytes.

Unlike the jump table approach of the previous section, in this method there is no default case and consequently identical blocks of code have to be repeated for distinct rejecting symbols. While this implies additional space to store the program, there is some gain in space in that a table mapping symbols to addresses does not have to be set up and stored.

Source-code 8. *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 it self
```

```
        mov dword [ebp-4], 0 ; first symbol of the alphabet
        mov dword [ebp+24],  alphabetSize
        mov eax, [ebp-4]
begin_loop:
        cmp eax, [ebp+24]
        je  end_loop
; record initial tsc
        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 jump address)
        jmp eax             ; jump to address
; specify the different case statements
case_0:
        mov edx,8
        jmp next
case_1:
        mov edx, -1
        jmp next
case_2:
        mov edx, -1
        jmp next
case_3:
        mov edx, -1
        jmp next
case_4:
```

```
            mov edx, -1
            jmp next
case_5:
            mov edx, 10
            jmp next
...
next:
          ;record final tsc
          ; calculate time
...
end_loop:
...
```

5.5 Data Collection

The collection of data was straightforward. 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 (low-level versions). Then, running through each symbol of the alphabet (i.e. char in algorithm 5 and 6) each hardcoded 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 that has 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, i.e. the number of accepting symbols, was used as a basis for clustering contiguous rows, resulting in 33 clusters of rows, containing 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. Refer to Appendix B for the reproduced data. 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.

5.6 summary of the chapter

In this chapter, we have given details relating to the implementation of the experiment that constitutes the cores aspect of the dissertation. Details about data collection have been presented. The next chapter covers details relating to the presentation of results.

Chapter 6

Experimental Results

6.1 Introduction:

Various graphical representations of data collected during the experiments described in the previous chapter are provided in this chapter. We start by presenting the processed results of the table-driven implementation. Then follows various graphical representations relating to results from both high-level and low-level hardcoded implementations. The chapter ends with a general graph to compare the table-driven implementation against the various hardcoded implementations.

6.2 Table-Driven Experimental Results

Figure 6.1 shows that the average processing speed of the table-driven method is about 88 clock cycles (ccs), both for accepting and rejecting symbols. The graph plots the first column (problem size) against the tenth column of Table B.1 of Appendix B. In this situation, the problem size does not appear to be a factor that influences the time taken to recognize a single symbol of some *FA*. Recall that the 88 ccs includes the time taken to perform a function (*fake_transition()*) call. In normal

circumstances the average time required to perform an assignment statement such as ($x = transition[i]$)is about 10ccs. Therefore, up to 78 ccs of the time provided in the figure is related to function call overhead and not the actual assignment statement.

The overall picture does not change if the time measurements for processing accepted symbols or rejected symbols is plotted. There is almost no difference between the time required to accept a symbol and the time required to reject a symbol. This is explained by the fact that access and retrieval of data from each entry of the table is performed at an almost uniform speed.

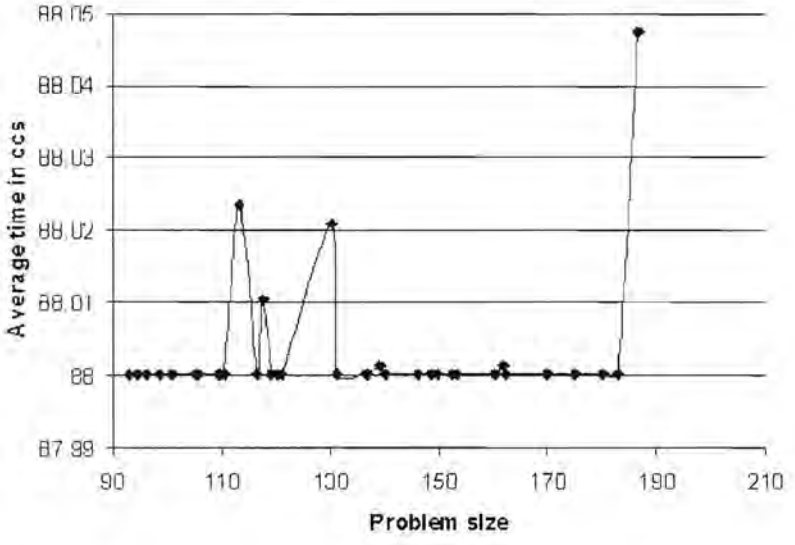

Figure 6.1: Average processing speed for Table-driven implementation (accepting and rejecting symbols)

6.3 Hardcoding Experimental Results

As mentioned in the previous chapters, there are several ways in which hardcoding could be implemented and five specific strategies have been selected in this work. The

reasons for various selections were previously explained. For presentation purposes, it was considered useful to partition the experiments into two groups: high-level language experiments and low-level language experiments.

6.3.1 High-Level Language Hardcoding

The encoding using nested conditional statements (*NCSs*) and switch statements (*SSs*) was considered for this group of experiments. Data collected for both methods are depicted in table B.2 and B.3 of Appendix B.

As previously suggested, there is a *prima facie* case to be made for anticipating that the *NCS* implementation may be influenced by the problem size (that is, by the number of symbols being processed)[1]. The reason was that comparisons are made sequentially from the top to the bottom of the control structure.

Figures 6.2 and 6.3 justify the arguments. When measurements are based on accepting symbols (*ASs*), then the average processing time is between 89 ccs and 100 ccs. For rejecting symbols (*RSs*), the average processing time is between 110 ccs and 160 ccs. This means that the processing of an accepting symbols is about 0.8 to 1.6 times faster than the processing of rejecting symbols. On average, therefore, rejecting symbols apparently require more processing time than accepting symbols. Furthermore, the average time for processing both accepting and rejecting symbols appears to increase with the problem size. In the figure, there appears to be a growth trend in processing speed as the the problem size increases. This can be explained by the increase, on average, in the number of sequential comparisons to be performed before accepting or rejecting a symbol.

Figures 6.4 and 6.5 plot the processing speeds obtained from the *SSs* implementation against problem size. No obvious trend is apparent in either of the figures – in

[1]If there were more states to be processed, then what is being termed here the "problem size" would translate to the overall density of the *FA*'s transition matrix.

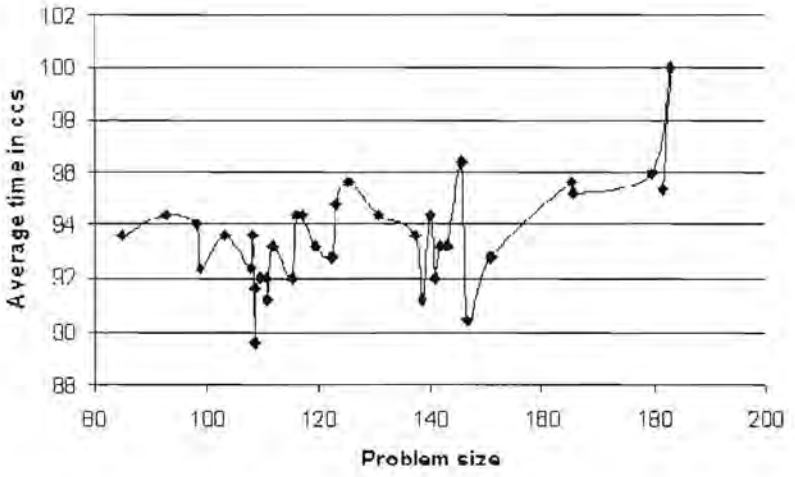

Figure 6.2: Accepting symbol performance for *NCSs*

both cases the processing time seems to oscillate unpredictably in a relatively narrow band around the average processing time. In the case of rejecting symbols, this average time is approximately 88.4 ccs whereas the average time for accepting symbols is some 92.4 ccs. There therefore appears to be a net processing speed difference of about 4 ccs between accepting and rejecting symbols.

A possible explanation for this difference might relate to the way in which the compiler executes the default cases. It would seem that these default cases are handled before any other cases. However, in the context of the current work, further investigations into this matter were not carried out, since the issue is not considered to be central to the main theme of the present work.

Having hypothesized on the basis of the above evidence that the problem size is not of importance at this stage of the experiment, it was decided to cross-compare the *NCSs* and *SSs* implementations on the basis of their average performance. Table B.7 in appendix B contains data reproduced by averaging the overall processing speed of both methods and other implementation techniques used for the single symbol

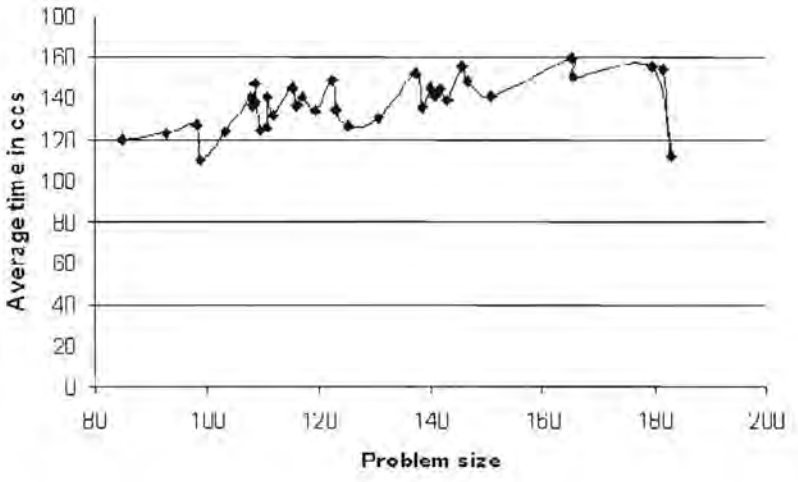

Figure 6.3: Rejecting symbol performance for *NCSs*

experiments. Recall that the data averaged (minimum, maximum and average time) was performed on the 355 data collected for each implementation technique, without taking into consideration the problem size. Figure 6.6 depicts in histogram form the overall performance of both methods – i.e. the average performance taken over all problem sizes. It appears that for minimum time measurements, the average time taken by the two methods is almost the same. In the worst case measurements (maximum time), the nested conditional statement approach tends to be slower than the switch statement approach. On average, the switch statements' average time is 90 ccs, which is about 33% faster than the nested conditional statements' average time of 120 ccs.

6.3.2 Low-Level Language Hardcoding

Jump table (JT), linear search (LS) and direct jump (DJ) were the low-level implementation techniques chosen, for the reasons previously described.

Data collected for the JT implementation is presented in table B.4 of Appendix

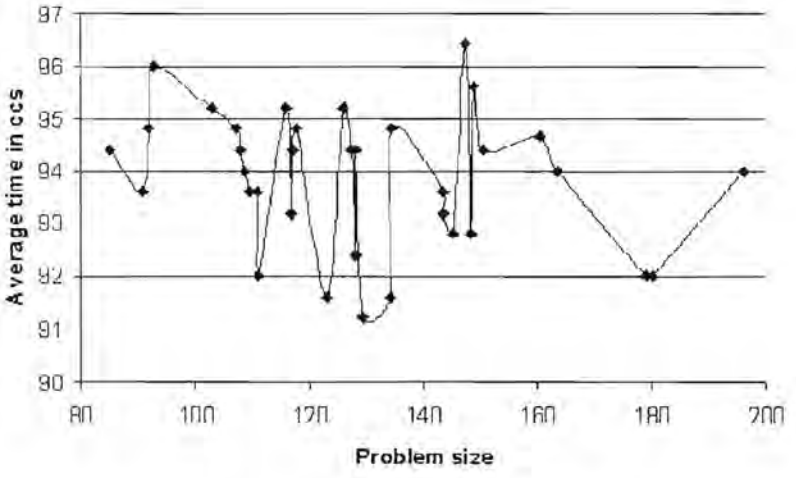

Figure 6.4: Performance based on *ASs* for *SSs*

B. Figures 6.7 and 6.8 depict the performances measured for both accepting symbols and rejecting symbols. Accepting symbols require an average processing speed of 8.5 ccs, whereas the average processing speed for rejecting symbols is about 4 ccs. Rejecting symbols' processing is therefore about 2 times faster than the processing of accepting symbols. In neither case, however, does the problem size appear to have any influence. The stable performance with respect to rejecting symbols may be explained by the fact that each entry in the jump table for a rejecting symbol executes the same default block of code.

In the case of the *LS* implementation, the problem size predictably constitutes a major factor that influences the time required to accept or reject a symbol. This is due to the nature of the structure that requires several sequential comparisons for a given symbol. As the problem size grows, processing both accepting and rejecting symbols may be expected to be slower since, on average, the number of comparisons required increases.

The data collected for *LS* is shown in table B.5 of Appendix B. Inspection of

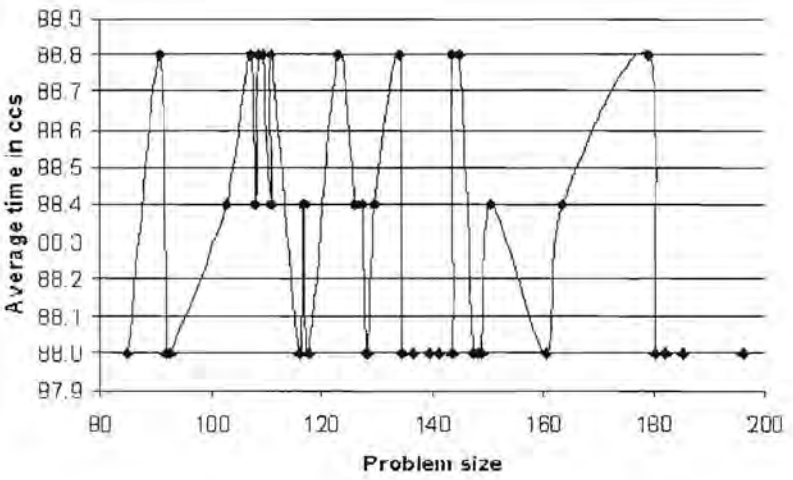

Figure 6.5: Performance based on *RSs* for *SSs*

figures 6.9 and 6.10 reveals that the method requires between about 2 and 14 ccs for accepting symbols, and between about 2.64 to 100 ccs for rejecting symbols. Rejecting symbols recognition is therefore up to about 10 times slower than the recognition of an accepting symbol. In accordance with the previously mentioned predictions, there is some suggestion of an upward trend in the graph for accepting symbols, and a very definite trend in the case of rejecting symbols.

Table B.6 of Appendix B contains the data collected for the *DJ* implementation. The method's processing speed does not appear to be affected by the problem size, as shown in figures 6.11 and 6.12. In effect, the minimum time taken to accept a symbol is between 12 and 18 ccs, whereas about 12 and 14 ccs are required to reject a symbol. There is thus a weak suggestion that the processing speed for accepting symbols is slightly slower than the processing speed for rejecting symbols.

Finally, as shown in figure 6.14, hardcoding implementations using low-level languages results in very fast minimum and average processing times. In fact, the minimum and average processing speed is in the range of 4 to 24.5 ccs. However, there are

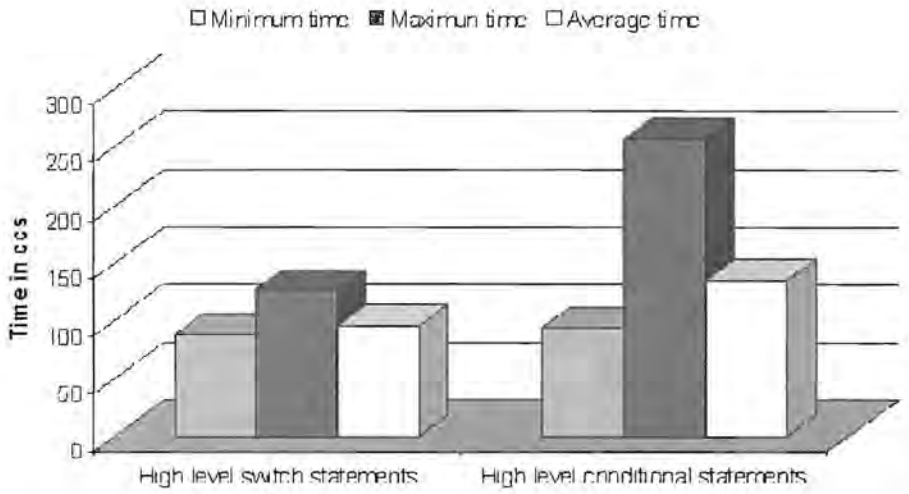

Figure 6.6: Performance based on hardcoding implementation in high-level language

differences between each of the strategies. If average or minimum time is considered as the basic measurement for comparison, then the use of a jump table appears to be the best of all three strategies. At a minimum, it requires 4 ccs to recognize a symbol, and on average it requires 13.3 ccs.

It should be noted that the average time data does not include outlier values, as mentioned previously. The measurement therefore does not represent an objective basis for comparison in the sense that the determination of what constitutes an outlier value was somewhat arbitrary. The intention was to eliminate observations from the sample data that did not accurately reflect processing speed variations due to random noise, but whose overly large values could more reasonably be ascribed to some chronic operating system factors that did not relate in any meaningful way to the present experiment – i.e. it is postulated that such factors occur, irrespective of the experimental task being run. In determining what values to consider as outliers, the rather arbitrary value of 100 ccs was chosen by inspection as the cutoff value.

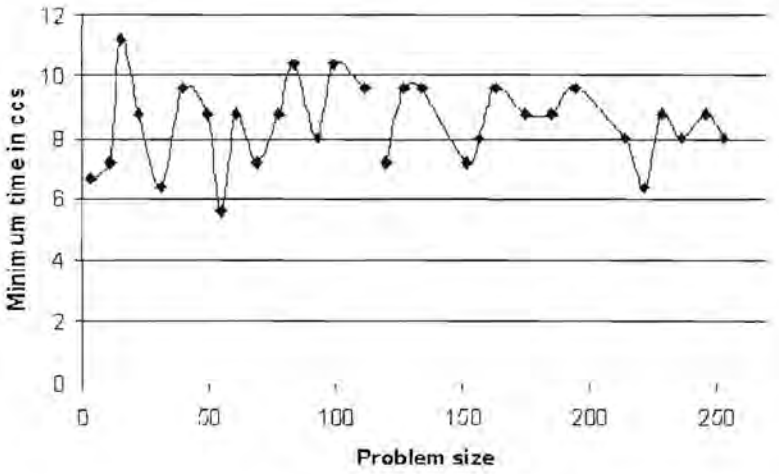

Figure 6.7: Performance based on *ASs* for *JT*

As a result, it is possible that several values that should have been considered to be outliers might remain in the averaged data. Nevertheless, there does not seem to be any obvious way of overcoming this difficulty.

To indicate the extent of these outlier values, the figures also show the maximum processing times. The fact that direct jump has the lowest maximum processing speed of 237.1 ccs is considered to be purely incidental. For reasons already mentioned, it does not seem like a reasonable metric for assessing the performance of the various implementation strategies.

6.3.3 Overall Results of Hardcoding

Figure 6.14 depicts the minimum, maximum and average processing speed, averaged over all problem sizes, for the various hardcoded implementations. The corresponding data may be found in table B.7 of Appendix B. The graphs strongly suggest that, in terms of this metric, the jump table implementation is the best of all the various approaches. It appears to be at least twice as fast as the linear search approach, at

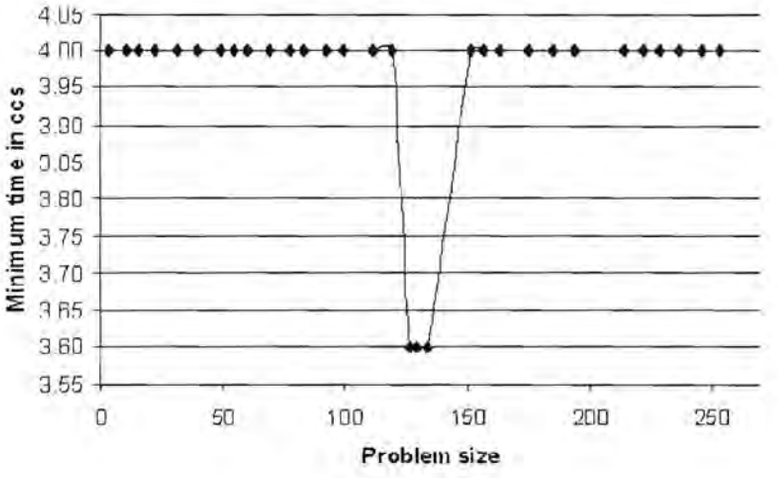

Figure 6.8: Performance based on *RSs* for *JT*

least four times faster than direct jump, and more than thirty times faster than either of the high-level hardcoded implementations.

6.4 Final Results

Investigation of the rest of the data revealed that, with the exception of the graphs characterizing the *linear search* and *nested conditional statements* versions, 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 6.15. The figure relates to overall average performance (as opposed to overall minimum or

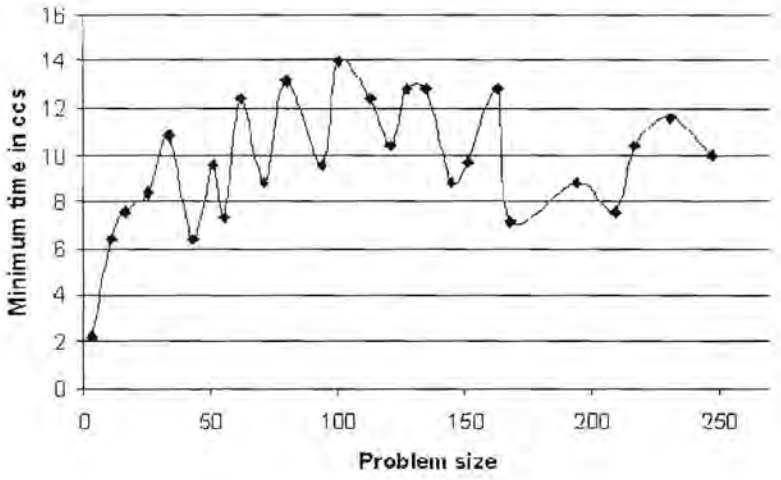

Figure 6.9: Performance based on *ASs* for *LS*

maximum performance) in regard to any symbol (whether accepting or rejecting). It follows that, 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. Moreover, it would seem that hardcoding to a high-level language is not worthwhile, since the standard table-driven implementation appears to be slightly faster.

6.5 Summary of the chapter

In this chapter, we have presented various results obtained from the first level experiments introduced in chapter 5. Table-driven and hardcoded algorithms were cross-compared and various graphs drawn. At this stage, it is shown that hardcoded algorithms are more time efficient that their table-driven counterpart. However, the traditional table-driven appears to be more efficient than the hardcoded algorithms

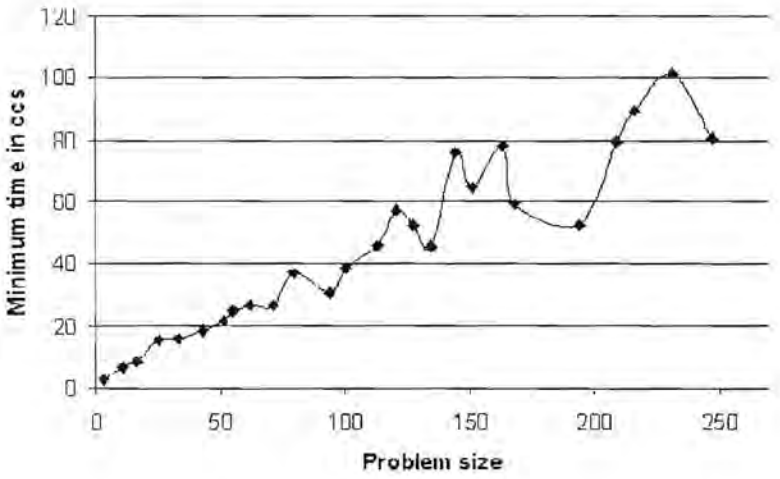

Figure 6.10: Performance based on *RSs* for *LS*

implemented in high-level language. The jump table implementation in a low-level language is the fastest of all other methods. However, it might be possible to obtain improved results with the direct jump implementation by engaging in further optimization.

Having established the efficiency of hardcoding implementations based on processing in regard to recognizing a single symbol in a single arbitrary state of an *FA*, it was considered desirable to scale up the problem to a higher level – i.e. to carry out experiments based on the recognition of an entire string within some arbitrary *FA*. Based on previously mentioned considerations, the overall expectation would be to obtain a linear time relationship with respect to the length of the string to be recognized by the *FA*. However, consideration must also be given to the overall cache misses encountered during the execution of such programs. The effect of cache misses introduces noise into the prediction of linear time performance. The next chapter presents in detail the problem and the results achieved thus far.

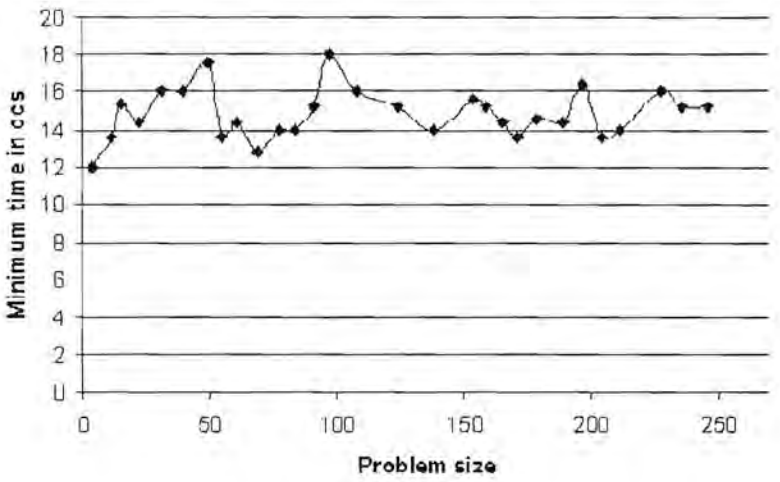

Figure 6.11: Performance based on *ASs* for *DJ*

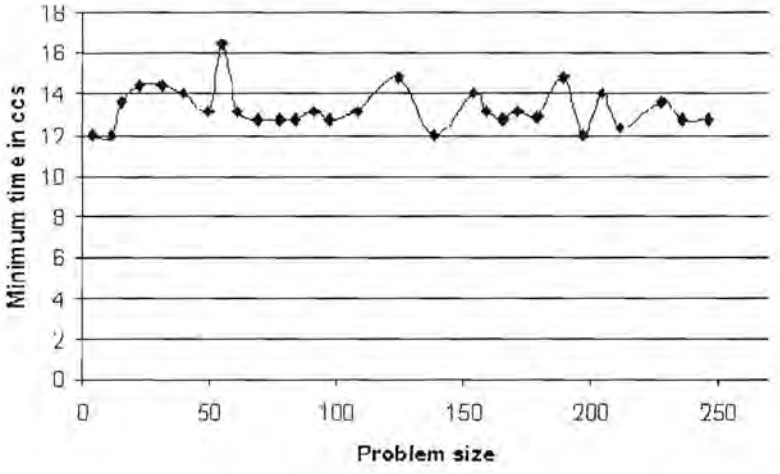

Figure 6.12: Performance based on *RSs* for *DJ*

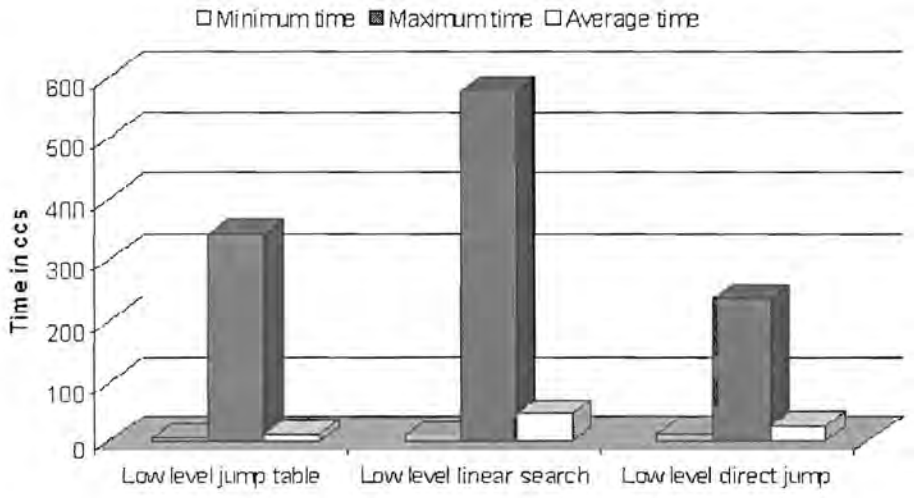

Figure 6.13: Performance of low-level hardcoded implementations

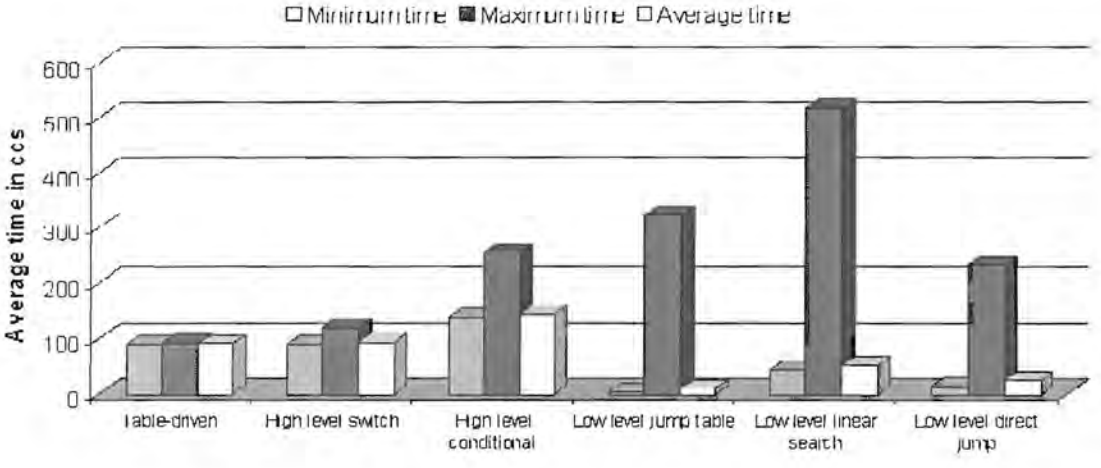

Figure 6.14: Performance based on hardcoding implementation

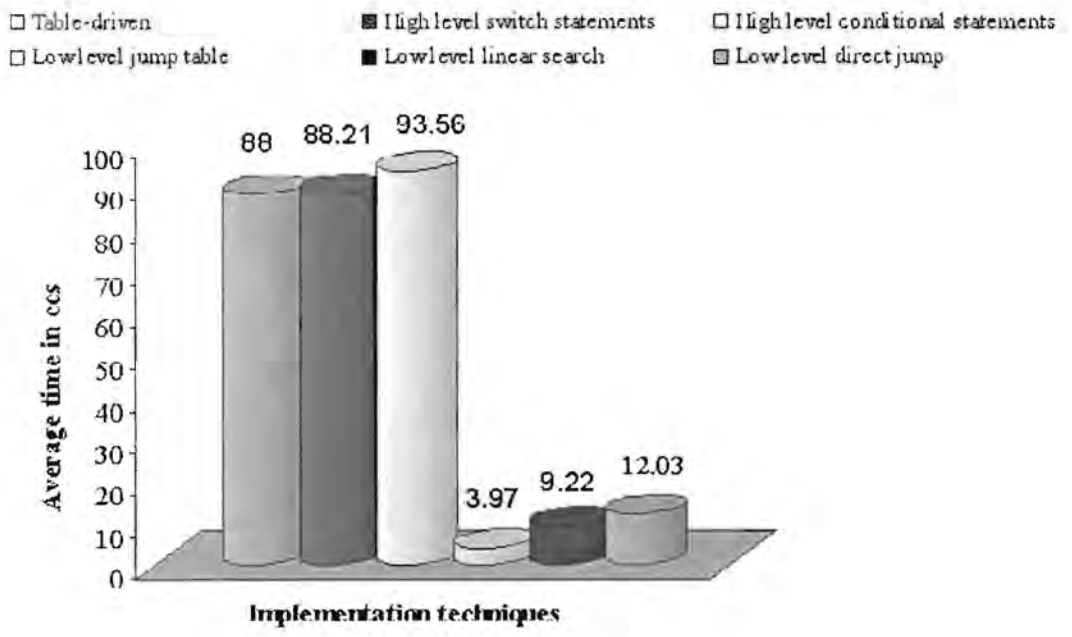

Figure 6.15: Average processing speed per implementation technique

Chapter 7

String Recognition Experiments

7.1 Introduction:

In this chapter, we extend the results obtained from previous experiments by performing quantitative analysis based on string recognition, where strings are no longer limited to a single symbol in length. One of the aims of this chapter is to investigate the effect of cache misses due to the size of the automaton being analyzed. It is shown that, within the range of experiments described below, caching effects could be detected as an influence on the processing time of hardcoded algorithms. However, variations in cache memory requirements did not appear to be required for comparable table-driven experiments. Extended experiments on string recognition processing were conducted using both methods. Below we describe the data that was collected, as well as the results and conclusions that may be drawn.

7.2 Exercising Memory on Intel Pentium Architecture

The implementation of a hardcoded *string* recognizer will clearly require a larger *FA* than generated previously. Because it implies an increase in the code size it may involve various levels of cache memory, main memory and virtual memory (through 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 in previous chapters. Unlike the table-driven algorithm, whose code size for string recognition is automaton-independent, the code size for a hardcoded program for string recognition is dependent on the automaton's features, i.e on the number of states, the size of the alphabet, the complexity of the transition function, etc. Memory management is therefore of concern, in so far as the program's processing speed is likely to be affected by its executable code size.

The amount of memory that a program can use is limited by the maximum address space of the processor [Ger02]. On Intel processors (Pentium 4 at 1GHz), the address space is comprised of physical memory and *virtual memory*. The use of *virtual memory* by an application is usually indirect in the sense that it occurs when the application requests more physical memory than is actually available. The process is referred to as *paging* and when used, the efficiency with which the application is processed is likely to be considerably degraded. Hence, programmers need ensure that – as far as possible – the program to be executed fits into physical memory.

Small and high-speed memories called *caches* are used to improve the latency of physical memory [Ger02]. The Pentium 4 processor has two caches called the L1 cache and the L2 cache. The L1 cache is split into two cache types referred to as a *trace-cache buffer* and a *data cache*, whereas the L2 cache is both a *data* and an *instruction cache*. The L1 trace-cache buffer can store up to 12-K decoded micro

operations in the order of program execution. It operates approximately at register speed, and is accessible in one processor cycle. The L1 data cache is relatively small (8KB) in size. It is always accessible within two processor cycles. The L2 cache is 32 times larger than the L1 data cache, but is about three times slower. It is therefore 256KB large and requires approximately six processor clock cycles to be accessed.

When an application accesses a piece of memory, no matter whether data is to be read or written, the processor first looks for the data in the L1 cache [1]. If found, a cache hit occurs and the data is accessed without interfering with L2. If not found, then there is an L1 cache miss. The data is then sought from the L2 cache. If found, there is an L2 cache hit. Otherwise an L2 cache miss occurs, and the requested data is not in either of the two cache memories. The data then has to be fetched from main memory. In general, instead of retrieving just the requested bytes, a 64-bytes chunk of data is fetched in the expectation that one or more of the remaining bytes are likely to be used shortly. In general, such a mechanism may, or may not increase the number of hits and therefore the efficiency of the application being processed.

The processing of an application can then be summarized as follows. The processor loads the program into the main memory if it is of reasonable size (i.e. if it can fit into main memory). Next begins a series of fetch/execute cycles as explained above. However, if the entire program cannot fit into main memory, there is a need for *virtual memory* and *paging* is performed by the operating system without any external intervention. *Paging* considerably reduces the efficiency of an application due to the *disk-access* nature of the operation. For the present study, paging does not constitute an issue since all programs that were written fit into memory and requiring intensive cache operations.

Based on all the above, one should clearly not anticipate for string recognition a

[1] Actually, the fetch is performed either from the L1 execution trace cache if the piece of memory accessed is an instruction, or from the L1 data cache if it is a data

simple linear-time scale up of the results obtained from the single symbol recognition case. Instead, the subsection below describes experiments for both table-driven and hardcoded algorithms that measure the effects of memory usage on the time efficiency of the various algorithms.

7.2.1 A Simple Experiment and Results

There are many dimensions in which arbitrarily large finite automata could be constructed and exercised: the number of states could be increased; the alphabet size could be made very large; the fan-out at every state could be increased (i.e. the density of the transition matrix could be increased); the length of input string to be recognized could be increased, etc. Since the current experiments were merely aimed at ensuring that large amounts of cache memory and main memory would be occupied, there did not appear to be any particular advantage in generating large automata that were unnecessarily complex. Instead, it was decided to generate large but fairly simple automata along the following lines.

The experiment was based on a language that has only two symbols in its alphabet (say a and b). For any state of the *FA* (except the final one), an input of a always triggers a transition to a next state, while b is a rejecting symbol in that state. 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). 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 before the $(n-1)^{st}$ occurrence of a would be rejected sooner. Conversely, any string with $(n-1)$ initial occurrences of a followed by an arbitrary number of a's and b's would be processed in the same amount of time, the outcome being to reject the string as being in the *FA*'s language after the first $(n-1)$ elements have been processed.

The advantage of using such a simple automaton is that the size can easily be

increased in a linear fashion (with respect to the number of states, and therefore also with respect to the size of the very simple transition table), and that each increase in size can easily be reflected in new hardcode that is the result of a relatively simple adaptation of the previously tested hardcode. This simplicity therefore supported the ability to create a relatively simple loop in which the size of the automaton is incrementally increased, string recognition measurements are made and the loop is then repeated.

The Hardcoded Experiment

A hardcoded implementation could be obtained by selecting the assembly source code of one of the hardcoded implementations previously described and then appropriately linking together n replications of this source code. For the present experiment, it was decided to use the best performing algorithm that had been previously tested, namely the algorithm based on a *jump table*.

Using the same approach as described in [KWK03], 400 different *FAs* were generated, with the number of states ranging from 10 to 4000. 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, giving the time to process the maximum length string, was recorded. The minimum, maximum and average values of these measurements over the 20 runs were calculated.

The Intel processor (Pentium 4 at 1GHz) uses the Branch Prediction Buffer (BPB) mechanism to make fairly naïve guesses (predictions) about the next path in a branching structure to take. The processor then executes some of the instructions in that branch in a pipelined fashion while simultaneously verifying that the branch is indeed the correct one to execute. If it guessed correctly, then some time has been gained. If not then it has to execute the instruction required by the alternative path, resulting to some latencies. The 20 runs performed in our experiments are implemented

using a loop, and are thus subject to branch prediction buffering. Furthermore, each iteration contains several branching instructions for both table-driven and hardcoded implementation. The following forms of branching structures are encountered by the processor:

- *Conditional branches executed for the first time.* An example of such a branching structure may be the *for loops* used for the table-driven implementation and various *je, ja* instructions (jump if equal and jump if above) used for the hardcoded implementation. This form of branching appears to be expensive during the first attempt by the processor to enter the loop. In general the first attempt by the processor will result on a mis-prediction since the actual content of the pipeline (Branch History Buffer – BHB –) is not made up of the desired instructions to be executed. Of course, a mis-prediction results in some latencies.

- *Conditional branches that have been executed more than once.* For such structures, there is a chance that the processor makes a correct prediction since the result of the previous branch is still in the processor's BHB. In that case, some processing time is gained.

- *Indirect jumps(jump tables).* For this kind of branch, the processor usually predicts that the branch target will be the same address as the last time the branch was encountered. This is likely lead to frequent mis-predictions if – as it is the case for our hardcoded implementation – the code not only has a large number of potential targets, but the same target is also seldom invoked twice in succession.

Details about how the processor handles various other types of branches using branch prediction buffering may be found in [Ger02].

The jump table used for the current experiment was rather simple (it contained only two entries corresponding to the two alphabet symbols). This suggests that the processor's BHB will contain accurate results during the entire process of recognizing the string. In the present context there is therefore a very low risk of mis-predictions. However, even though the current experiment's implementation was very simple, it should be noted that noisy time measurements are likely to be obtained in other more elaborate hardcoded experiments since the hardcoded implementation is based on *indirect jumps*.

As a lower limit for the time taken to process a single symbol in an entire string, it seems reasonable to rely on the previous results obtained[2]. Consequently the minimum value was used for processing a single symbol as determined by the jump table experiments described in previous chapters. The ideal, excluding the effects of different levels of memory, would be a linear time scale-up proportional to this lower limit and proportional to the length of the input string. Put differently, any time measurement not reflecting this proportionality could be attributed to caching effects.

The processing speed (minimum, maximum, and average) for various hardcoded multiple state programs was recorded. We then divided the respective processing speeds for each algorithm's execution by the number of states in that experiment, to obtain the average minimum speed per state (or, equivalently in the present case, per input symbol) for automata of various sizes. The series of data that was produced was plotted and cross compared with the jump table hardcoded experiment. Figure 7.1 depicts the resulting graph.

The results show a fairly constant growth rate in certain regions, then changes to

[2]Note that the lower limit for the table-driven case excludes the time taken to switch contexts in order to execute the function *fake_transition()* as well as the time taken to return from that switched context.

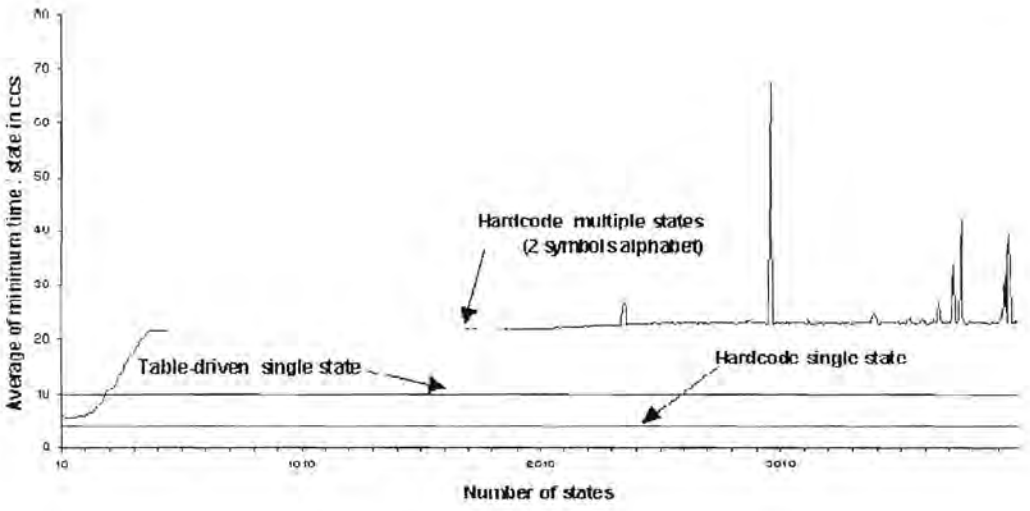

Figure 7.1: Hardcoded time against automaton size for two symbols alphabet

a linear growth rate until another plateau of constant growth rate is reached. The overall pattern is plausibly explained by the following:

- The size of the L1 cache – the trace cache –(12KB) cannot contain the code for even the smallest automaton (10 states) whose executable was measured to be approximately 16.09 KB. The program size is therefore bigger than the L1 cache's size. Consequently, services of the L2 cache (256KB) are required for complete processing. But between 10 and 110 states, the L1 hit rate is fairly hight, and the need for L2 accesses is correspondingly low. As a result, the average time per state is fairly constant for these automata.

- Between about 160 and 360 states, the need for the L2 cache by the processor increases progressively and so does the probability of cache misses at level L1. As a consequence, the average processing time per state shows a linear increase in this region.

- An almost constant time per state is experienced from between about 460 and 1700 states. The associated hardcode executable appears to be able to fit into the L2 cache. This is illustrated by the smoothness of the curve within that range.

- From about 1800 states onward, the minimum size of the executable is approximately 260, and this can no longer fit into the L2 cache. A very slow but linear growth in the curve was observed. This indicates that services from the main memory were required even though hit rates in the L2 cache still remained relatively high in the range of states exercised by this experiment. Furthermore the policy of moving across 64-bytes of data from main memory at a time, coupled with the relatively simple structure of the *FAs* under test probably contributed to the very small growth in average processing time per state, barely visible in this region of the graph. Note that the largest number of states tested in the experiment was 4000 and the corresponding code size was 561 KB. This is approximately twice as large as the L2 cache size. Consequently, we may expect that beyond 4000 states, similar effects that were seen in the case of the L1 cache above will be observed for the L2 cache – i.e. an increase in the number of misses in L2, and an increase in the number of hits in the main memory. Such observation was explained in detail in [KWK03].

- In the figure, noise is observed between 2310 states and 2410, 2910 states and 3010, and from 3610 onward. These phenomena are possibly explained by the fact that the processor's BHB may contain inaccurate results during the branch

prediction buffering. The nature of the indirect branching structure of the experiment explains such a situation. As a result, mis-predictions that cause a considerable waste of time to the overall process are observed. However, further experiments is needed to fully justify this claim.

- We may expect that in the long run, if the main memory is full, that paging between memory and hard drive will be carried out. However, the effects of paging were not further investigated in this study.

The Table-Driven Experiment

The table-driven algorithm was easily constructed using algorithm 3 from chapter 3. The same 400 *FAs* that had been generated in the above hardcode experiments were used in this experiment. Again 20 runs were performed per automaton, in each case determining the minimum, maximum and average time to recognize a string of length n, where n ranged from 10 to 3999 (and corresponded to the total number of states minus 1 in the respective automata).

The results of experiments based on repeated runs of the table-driven algorithm are depicted in figure 7.2. The data again refers to the average of minima over the 20 runs per automaton.

The figure clearly shows that no cache effect is experienced for the table-driven method, but rather that the memory load appears to be the major effect. The average time taken for processing is slightly above 40 ccs, which is considerably more than the time measured when processing a single state. This is justified by the fact that more statements per state are executed in the present compared to the single-state experiment.

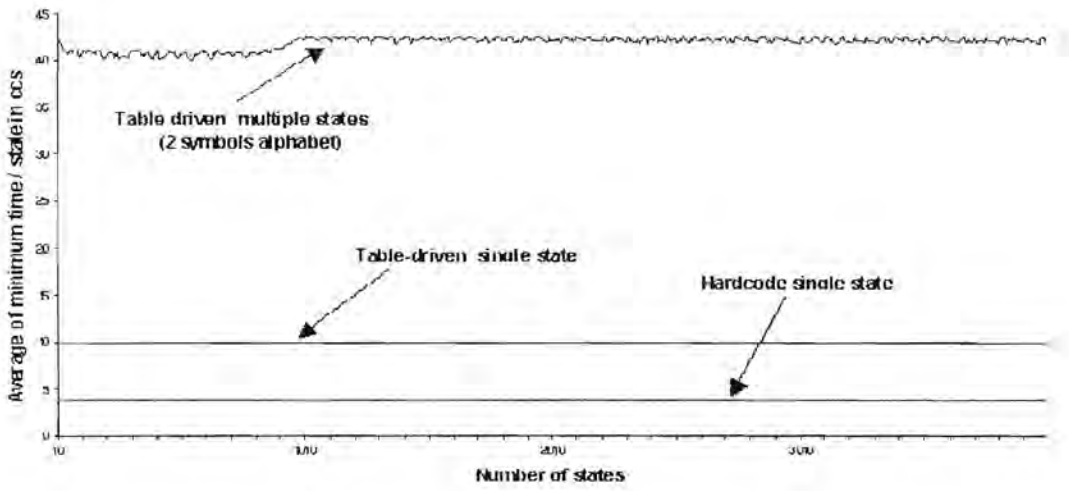

Figure 7.2: Table-driven time against automaton size for two symbols alphabet

Putting it all together, figure 7.3 depicts the general performance of both hard-coded and table-driven implementation based on the simple two-symbol alphabet scenario. It clearly shows that the hardcoded algorithm outperforms table-driven implementation. These results are described in [KWK03].

However, it would be naïve to consider these results as fully representative since the primary objective of the experiment was to establish and observe the effect of cache memory on the hardcoded algorithm. The next section discusses a more realistic set of experiments based on a set of more realistic string recognition problems. This will enable more accurate conclusions to be drawn.

7.3 The String Recognition Experiment

Having observed and plausibly explained the cache effects from the previous section it was necessary to perform a more realistic experiment to more accurately compare the table-driven and hardcoding finite automata processing. The idea was first of

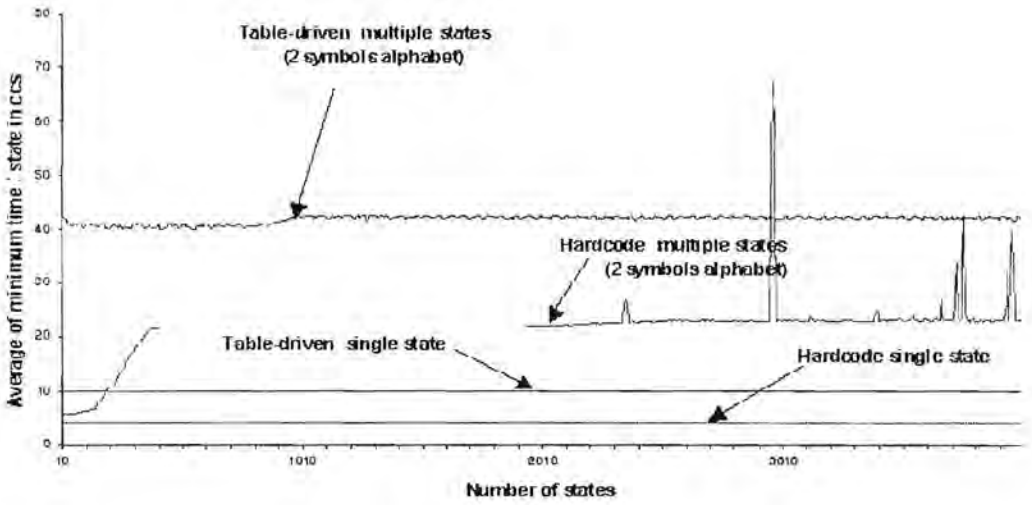

Figure 7.3: table-driven and hardcoded multiple states for two symbols alphabet

all to randomly generate a finite number of transition tables and their corresponding hardcoded executables. These generated programs were then executed and required times to recognize a string were recorded. The time taken to recognize the same string based on a table-driven algorithm was also recorded.

In general, there are two ways of implementing a string recognizer: implementation using symbol searching and implementation using direct indexing. Both methods are briefly explained as follows:

- The implementation using searching refers to a situation where the alphabet set is explicitly required when processing the string that is being recognized. An array is used to store the symbols of the alphabet. The column indices in the transition table correspond to the indices in the alphabet array, thus indicating which alphabet symbol the column is intended to represent. Therefore, during the recognition process of the current symbol, it is necessary to *search* for its position in the array of symbols before accessing the entry in the table using

the assignment

$$nextState := transition[i][pos(str[stringPos], alphabetArray)]$$

of algorithm 1. Of course, *pos(ch,array)* returns an integer value that reflects the index of the symbol in the array of symbols (alphabet).

- The implementation using direct indexing is rather straightforward. No search is required to access an entry in the transition table. The position of the symbol to access in the transition array is known, and is indeed the symbol itself[3]. Therefore the assignment

$$nextState := transition[i][str[stringPos]]$$

refers to an entry in the table, and will return either -1 if there is no transition and a non negative integer otherwise.

The first method is, not surprisingly, more time consuming than the second. The time may be optimized if a binary search algorithm were to be used to search for the character's position instead of a linear search. However, since that would have made the assembly language coding in the hardcoded case slightly more complicated, the present experiment simply stuck to linear search. For the second approach, the conversion from a character data type to an integer data type will undoubtedly take a certain amount of time. However, we did not take such factors in consideration during the experiments, since it was assumed that the approach is used with the standard type conversion from *char* to *int* and vice versa offered by most programming languages.

[3]This may be regarded as a character symbol, whose value can be printed, and its conversion to integer type represents its index in the transition table.

The experiment for both approaches was based on the random generation of 400 different automata (transition tables) but always using a 10-symbol alphabet, say {a,b,c,d,e,f,g,h,i,j}. The number of states of the automata were varied between the ranges of 10 to 4000 states. For each automaton of size n, a string of length $n-1$ was randomly generated, but always in a way that the entire string would be recognized by the automaton. We call such a string an *accepting* string. The filling density of each transition table was set at 41%. The subsection below presents the results obtained for both methods.

7.3.1 Experimental Results

Experiments on string recognition, where a linear search algorithm was used to retrieve the position of the current symbol from the alphabet symbol array, is depicted in figure 7.4. The data used to reproduce the graphs can be found in Table B.9 of Appendix B. The figure clearly indicates the effects of caching for the hardcoded experiment, and memory load for the table-driven experiment. An important observation to make here is the effect of L1 cache on both table-driven and hardcoded implementation. In fact, for very small automaton, say ones having at most about 50 states, one notices that the L1 trace cache has an impact on the hardcoded implementation whereas the L1 data cache influence the table-driven approach. This is apparent in figure 7.4 by the portion of the graph just below 200 ccs for the table-driven method (L1 data cache effects), and the portion of the graph below 175 ccs for the hardcoded implementation. We observe that the hardcoded algorithm outperforms its table-driven counterpart for automata of size less than approximately 2000 states. Above about 2000 states, the table-driven experiment appears to be the better method to use: it requires a constant time of about 200 clock cycles for processing while its hardcoded counterpart becomes increasingly inefficient with a processing speed of up to 300 clock cycles. The inefficiency of the hardcoded methods above

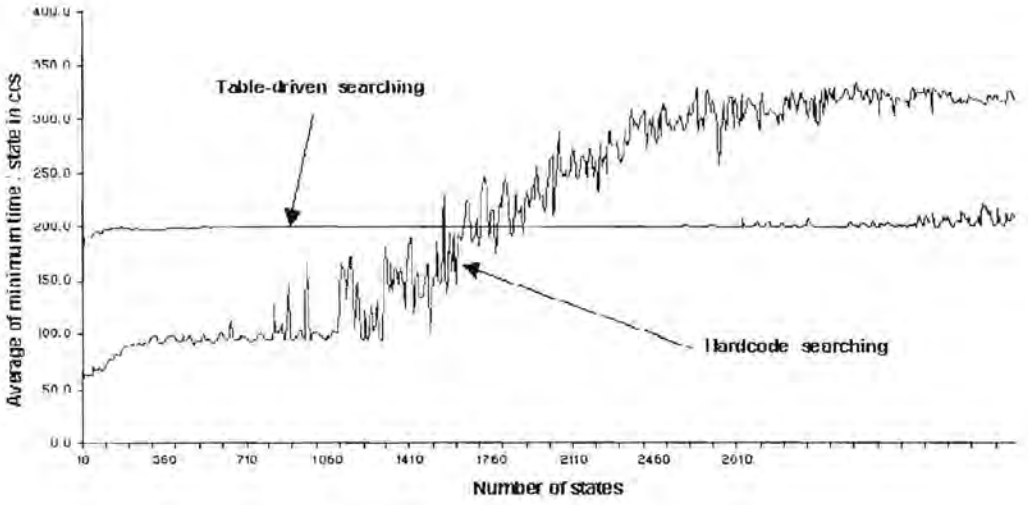

Figure 7.4: Table-driven and hardcoded performance using linear search

that threshold is justified by the same cache effects as previously discussed in the 2-alphabet experiment. However, the number of noise increases because the *FA* is less regular than in the two-alphabet case.

As an aside,note that if a binary search algorithm had been implemented instead of a linear search, a global improvement would be observed on the overall results in both cases. (The binary search time complexity is logarithmic compared to the linear search which is, of course, linear.) If the binary search algorithm was encoded in the table-driven implementation only, then the competitiveness between the techniques would have been reduced at less than 2000 states. However, if both methods rely on a binary search, one could expect that the hardcoded implementation would continue to outperform its table-driven counterpart at above 2000 states.

Table B.9 of Appendix B provides the data collected in the experiment based on the direct indexing of symbols. Figure 7.5 and 7.6 depict the performance of both methods. The figures clearly show that the hardcoded implementation is of interest

when using automata of no more than 500 states. However a closer look on the graphs in figure 7.6 shows that between about 500 and 1500 states, the two methods compete. By this is meant neither method is a clear and consistent winner within that range. From 1510 states onward, however, the table-driven starts to become the more efficient approach, continuing to have a constant processing speed of approximately 45 clock cycles per state.

An observation of interest is the difference in speed in the implementations using direct indexing and the implementations relying on searching. Figure 7.7 shows the gap between the two table-driven methods, as well as between the two hardcoded algorithms. The table-driven implementation using searching requires approximately 200 clock cycles per state, while its direct-indexing counterpart requires only about 45 clock cycles per state – i.e. about 155 clock cycles difference between the two approaches. Of course, the gap observed might be reduced if a binary search were to be used. However, this was not tested as part of the current round of experiments.

A comparison between the two hardcoded graphs in the figure shows that for the first 900 states, the gap between the two approaches is almost constant at about 50 clock cycles. This gap is reduced as the number of states becomes larger and tends to be negligible above 3000 states. Such an observation clearly demonstrates how *instructions-dependant* a hardcoded algorithm appears to be. Therefore, no matter the complexity of the algorithm being implemented, the processing speed is *code-size* dependant. *The bigger the code, the more the processing speed.* Finally, when implementing *FAs*, one should take into account the number of states of the device being encoded. A rough rule of thumb would be to used the hardcoded approach for automata of up to about a 1000 states and the table-driven algorithm for automata above that limit. What should happen if the memory limit has been reached for table-driven experiment is a topic of separate interest.

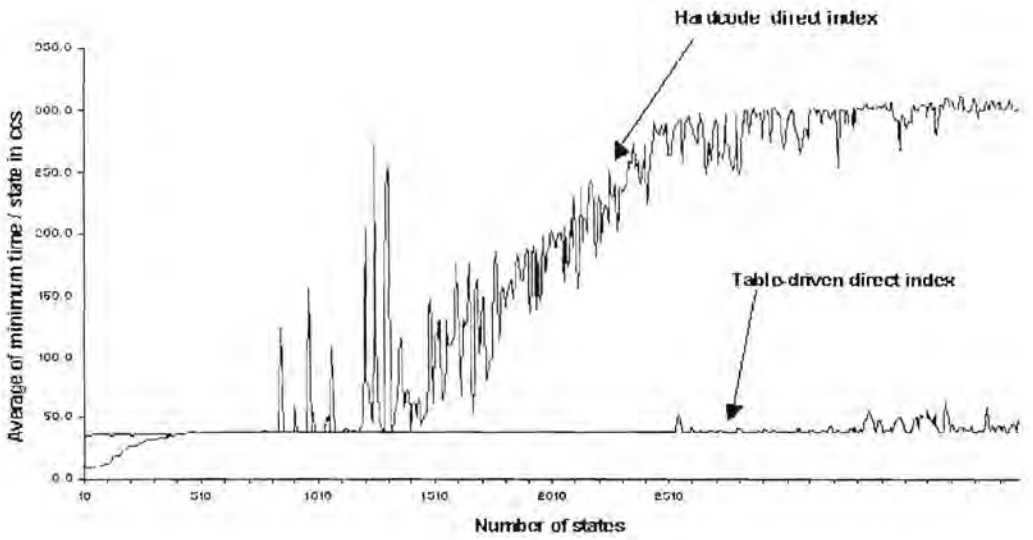

Figure 7.5: Table-driven and hardcoded performance using direct index

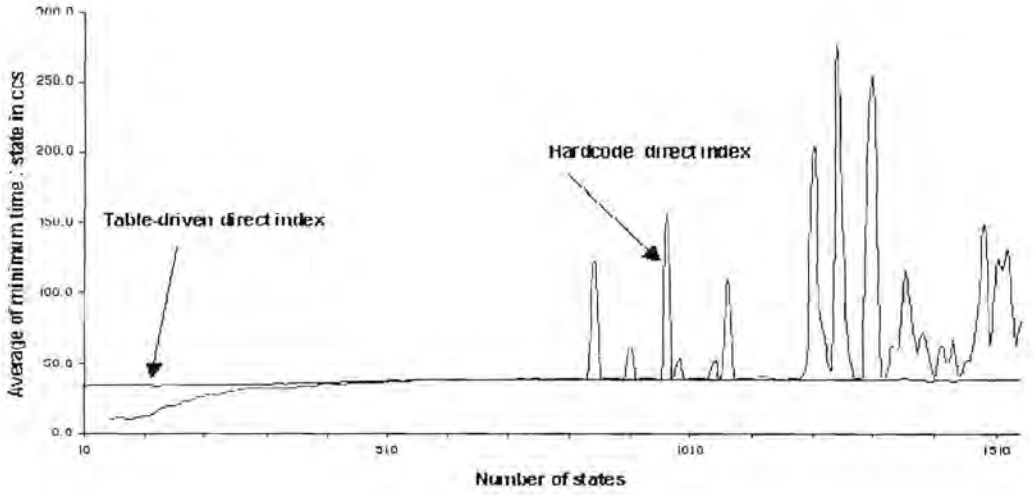

Figure 7.6: Table-driven and hardcoded performance using direct index

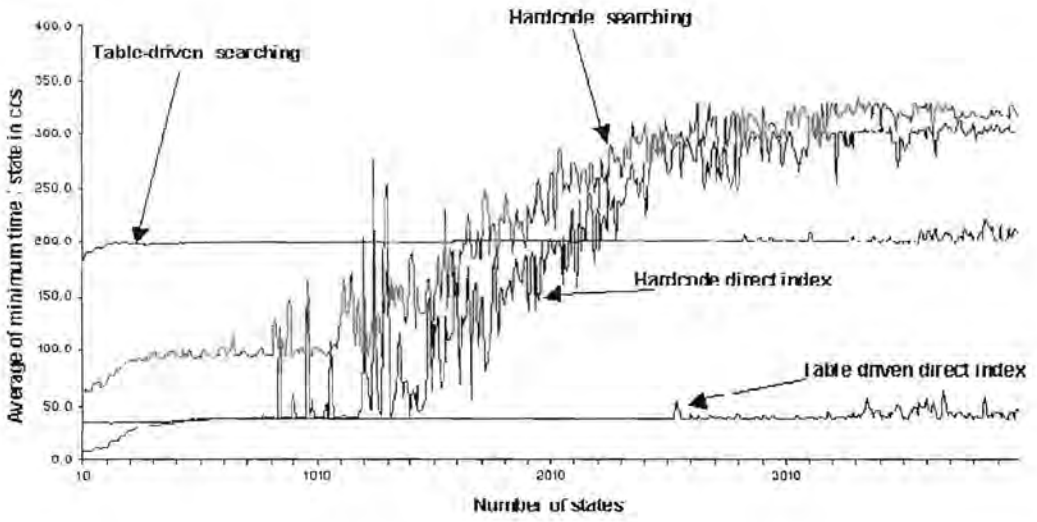

Figure 7.7: Performance based on searching and direct indexing

7.4 Summary of the Chapter

In this chapter, we have used a specific and fairly simple type of *FA* to perform some basic experiments on the recognition of a string by an automaton. This experiment made visible the effect of cache memory on the hardcoded implementation and strengthened the idea that table-driven implementation is memory load dependent[4]. We have also performed a more realistic experiment based on the recognition of strings from a somewhat larger alphabet than in the first experiment. We have shown that the hardcoded implementation provided very interesting processing advantages over it table-driven counterpart when implementing automata of up to about 1000 states in size. However, the table-driven method is preferable when the automaton to be implemented is above 1000 states (and, of course, the system has enough memory to avoid unfortunate situations such early program termination or system halting as

[4]By this we mean that if the processor's memory is not big enough to contain the actual table being implemented, important processing deficiencies will be observed

mentioned in chapter 1). One of the main issues that should be raised is that, if the system is limited in terms of memory, how could we overcome the situation using the hardcoded implementation? A discussion around this question will be presented in the next chapter for future work.

Chapter 8

Summary and Future Work

8.1 Summary and Conclusion

Hardcoding of finite state automata involves the design and implementation of an algorithm that does not require external data at run-time. All data that the algorithm needs is embedded in the algorithm itself. Our study was carried out in three major phases:

The first phase was the preliminary experiments based on single symbol recognition [KeWK03]. Several implementations were empirically investigated:

- *Nested conditional statements*: The transition function is translated into a set of nested conditional statements, (that is, if-then-else statements) which test whether to accept or reject a symbol. The testing is done on a sequential basis, meaning that the number of comparisons to be done may negatively affect the processing speed.

- *Switch statements*: The transition function is implemented in the form of a branching structure, such that control branches to statements relating to the

corresponding symbol and these statements either trigger the transition (if there is any), or execute the actions related to no transition. This method appears to be the best way of hardcoding the symbol recognition algorithm in a high level language.

- *Jump table*: The transition function is represented in a low-level language in the form of code that relies on a so- called jump table. Thus for each alphabet symbol an address is stored in the data segment that points to the label in the text segment at which instructions for dealing with that particular alphabet symbol commence. The relevant position in the table from which to retrieve this label's address, is computed from the value of the input symbol as well as from the start address of the table in the data segment.

- *Linear search*: This is an optimized low-level translation of nested conditional statements. It is implemented exactly in the same way. It is therefore subject to same limitations as its high-level counterpart, although the fact that it is implemented in assembly results in improved processing speed.

- *Direct jump*: For each input symbol, a direct jump is executed to an address at which code relating to that input symbol is stored. Thus instead of first referring to a jump table, the relevant code address is directly computed from the input symbol. Although this method seemed conceptually more efficient in terms of processing speed, this was not borne out empirically. Further optimizations to the existing code might be possible, but this is a matter for further study.

The second phase of our research was related to an empirical evaluation of caching effects in relation to the code-size of hardcoded algorithms [KWK03]. It was shown

that, while the table-driven implementation of *FAs*, heavily depends on the size of the transition matrix represented in the memory, the hardcoded implementation strictly depends on the number of instructions required to implement the transition. Therefore, the processing time required to recognize a string, is not only a function of the string's length, but is also a function of the size of the hardcode generated for the automaton implementation. If the hardcoded code is small enough to fit in cache memory, the processing speed will be highly efficient. However, if the hardcode is very large such that it cannot be contained in cache memory, the processing will be performed from main memory, and therefore results in increasing inefficiencies as the code-size grows. The jump table method was chosen a basis for the experiment since preliminary results showed that it was the best of the hardcoded methods that had been investigated. However, for future work it might be worthwhile to consider the caching effects that arise if a direct-jump implementation is used.

The last phase of our empirical study was based on a quantitative evaluation of fairly realistic string recognizer so that important recommendation can be drawn from our research. The study was based on an evaluation of the processing speed of *FAs* of various size (number of states). We found it reasonable to generate randomly an accepting string proportional to the size of the automaton, so that the minimum path through the automaton would always be traversed during string recognition. The results presented in the previous chapter showed that the hardcoded implementation of *FAs* seemed to be of high efficiency in terms of processing speed when implementing *FAs* of not more than a thousand states. For automata having more than a thousand states, the table-driven implementation tends to be of high efficiency. However, the forgoing needs to be qualified by noting that the experiment relied on an alphabet of size 10. These conclusions would have to be appropriately modified for larger alphabet sizes, by taking into account the fact that hardcode (and therefore speed) would accordingly increase.

The final experiment performed only considered strings proportional to the automaton size and the transition table density was held at 41%. In that regard, the following issues need to be highlight:

- *What if the automaton's transition table density is at 100%?*: This situation does not appear to be of great concern in the context of the current implementation choices. The number of entries in the transition table depends on the alphabet size and the number of states. Each entry is either a non-negative integer (a next state) or a negative value (-1 in the current implementations) to indicate that there is no transition. If the -1's were to be changed to non-negative integers, nothing would change in regard to the time required by the recognizer to accept or reject a string. The transition table density therefore does not affect the processing speed of either implementation approaches. Of course, these conclusions (especially in regard to the table-driven approach) would need to be modified if the transition table were to be implemented differently from the current study, e.g. as a structure of linked lists that does not explicitly store "no transition" information. However, these matters have been considered to be beyond the scope of the current study.

- *What if the length of the string varies (increases, decrease)?*: Varying the string length automatically varies the total processing speed required to accept or reject the string. However, it does not significantly vary the processing time per symbol string. Therefore for both table-driven and hardcoded implementation, the per-symbol timing recorded and comparisons will remain of the same magnitude as already presented in the results of the last section of chapter 7.

- *What if the alphabet size increase?*: The number of symbols, in the alphabet

constitutes an important factor for implementing both table-driven and hardcoded experiment. If the alphabet size increases, the size of the transition function increases. Therefore the memory load increases for table-driven implementation and the code size increase for hardcoded implementation. As a consequence, the efficiency of both table-driven and hardcoded experiment will be reduced in line with the caching and memory management effects that arise. At the time of writing, quantitative evaluations of such effects were not available. However, experiments are currently in progress and the results will be released outside the scope of this dissertation.

To date, experiments have been performed on randomly generated *FAs*. Results obtained are rather general and might not be applicable for certain specific types of *FAs*. Further experiments need to be carried out such that a sort of taxonomy of *FA* implementations can be built up, indicating which "families" of *FAs* would be best suited to which implementation strategies.

Moreover, it might be possible to fine tune both the hardcoded and table-driven implementations that have been produced thus far. This could yield better processing speeds than those provided by this work. We have restricted ourselves to using a jump-table for the hardcoded implementation. Only time-constraints on the present project prevented us from attempting further optimization on the direct-jump implementation and using it for the experiment. Such an approach might eventually produce better values than the ones obtained in this work. Moreover, it is also possible to combine the linear search, jump table and direct jump together to produce better performance than the one produced in this work. Another important issue is the fact that we did not explore the hardcoded implementation in a high level language. It will be of interest to empirically investigate this aspect of our work for further usages

In summary, the foregoing experiments have shown that there are certain occasions when a hardcoded solution to the problem of *FA*-based string recognition is more time-efficient than the classical table-driven solution. In the context of these particular experiments, the hardcoded solution appeared to be better than the traditional table-driven method for automata of about a thousand states. The table-driven algorithm outperforms the hardcoded counterpart for an automaton having more than a thousand states.

8.2 Future Work

While the results to date indicate that hardcoded technology outperforms the table-driven method in some contexts, it is desirable to perform more extensive quantitative experiments in various directions. The following issues should be considered in doing so:

- ***Problem size***. The fact that the size of the hardcode will grow linearly as the number of states becomes larger represents an important limitation on the overall processing performance for both approaches in the longer term.
- ***Decomposition of the automaton***: In reflecting on the above limitation, one might consider decomposing the problem into sub-problems – that is, partitioning and storing parts of the hardcode on different files;
- ***Compression***: It seems relevant to consider issues related to compression in order to save space.
- ***Paging***: As the problem size is enlarged, it seems relevant to investigate at which point paging between main memory and hard drive will begins to take

effect. The impact of such paging on both the table-driven algorithm as well as on the hardcoded versions should be explored.

- ***Different platforms***: The results to date all relate to Intel Pentium 4 architecture. It is desirable to verify empirically whether the overall results will remain the same on different hardware platforms.

- ***Hardcode generation tools***: Currently, much of the software that has been developed has been for research purposes. In the longer term, it would be useful to develop user-friendly hardcode generators that take an *FAs* language's specification as input (for example, in the form of a regular expression or as a right linear grammar) and directly produce the hardcode in executable form.

Once the above matters have been satisfactorily investigated, the hardcoded implementations might well be usefully employed in the various computing field that rely on *FAs* as a modelling tool. This is because, to our knowledge, hardcoding in the context of formal language theory and practice, has mainly been used to do parsing, and more recently, for programming regular expressions in C# [KIM02]. However, it would be reasonable to expect similar satisfactory results in several area where *FAs* are used for modelling purposes, such as biological computing, machine learning with neural networks, genetic algorithms, and many other domains where *FAs* could be used for modelling. A combination of modelling optimization and encoding optimization such as automata minimization and hardcoding should be encouraged as that would yield highly efficient solutions to the related string-recognition problems.