Part I

Prologue

CHAPTER 1

INTRODUCTION

Computational problem-solving often relies on modelling that is based on finite automata, and that subsequently requires the need to manipulate automata *symbolically* or *concretely*. By the symbolic manipulation of an automaton is meant here, the application of various classical operations on the automaton, thereby transforming it with the aim at producing another automaton for further usage. Examples of such symbolic manipulation include constructing the automaton from a regular expression, determinizing a non-deterministic automaton, minimizing an automaton, etc. On the other hand, concrete manipulation of an automaton is taken to mean processing based on the automaton's definition with a view to determining string ownership of its associated language. The concrete manipulation of automata is usually referred to as automata-based string processing, or more specifically automata-based string recognition. Various real-life problems in medicine, physics, chemistry, and even economics could be modelled using automata, and their solution subsequently requires symbolic and concrete manipulation of automata [Vic84, McC97, Deu99].

Some well-known problems that exploit concrete automata processing as part of their solution are applications such as network intrusion detection systems, DNA analyzers, tandem repeat finders, natural and computer virus scanning, spell checkers, virtual circuit simulation and many more. In those problems, an algorithm is used to test whether a given string is part of the language modelled by the automaton that is associated with the problem being solved. In general, the test of string ownership by a language is done using the conventional table-driven algorithm.

When testing whether a string is part of the language modelled by an automaton using the table-driven (TD) algorithm, the automaton is normally represented in memory in the form of a two-dimensional array. This array is then accessed by a driver function which scans each symbol of the string and tests whether it triggers a transition to a next state of the automaton. The table-driven algorithm is hampered by a *memory load* problem since as the automaton size grows, more memory in needed to represent the transition table.

An alternative to the TD algorithm is the so-called hardcoded (HC) algorithm, suggested by Thompson in the late 60's [Tho68], whereby simple instructions are used to represent the transition table. Here, string acceptance testing no longer relies on accessing data in a table in memory, but instead, on the direct execution of instructions that determine the next transition. As for the TD algorithm, the HC algorithm is hampered by the problem of instruction size, since as the table grows, more instructions are required to represent the whole transition matrix.

To the best of our knowledge, only the TD and HC algorithms are available in the literature for automata-based string processing. Moreover, the HC algorithm is proven to be more efficient than its TD counterpart when processing automata made of states in the order of hundreds [Nga03]. In theory, the performance of an automaton-based string recognizer is linear in the length of the string being tested for acceptance. However, various other factors such as the hardware capabilities, the automaton's number of states and even the automaton's alphabet size are non-negligible factors that play a role in the performance of automata-based string recognizers.

Previous investigations in [Nga03, NKW06a] revealed that the performance of both HC and TD algorithms are correlated to the hardware's cache memory. Thus, the performance of these two implementation approaches depends on the extent to which the cache memory has the capacity to hold enough data or instructions during string acceptance testing. It is therefore of interest to further investigate alternative algorithms that take into account the computer's cache memory capabilities. This will not only fill a gap in the literature (for further studies), but might also lead to algorithms that are better than existing ones in terms of performance.

1.1 The Problem

In practice, the performance of an automaton-based string recognizer may not always depend on the length of the string being tested for acceptance as it is the case in theory. Factors such as the automaton's alphabet size, its number of states and of course its layout (its sparsity and its density) are of importance when dealing with efficiency. On top of all these factors, is the computational medium to be used for processing. In effect, although the performance of any algorithm be it very efficient in theory, may in practice be hampered by the hardware being used for processing. Many factors in this context are a cause of concern, but the most prominent one in our opinion is the cache memory. Algorithms are likely to have their *performance boosted* if cache is taken into account when organizing data and instructions. It is based on such observations that we can summarize the whole problem to be addressed in this thesis as that of "**strategies for optimizing cache usage in finite automata-based string recognition**".

The work seeks to investigate and propose new automata-based string recognizers by taking into account data/instructions organization for efficient cache usage. In practice, the two-dimensional array used in the table-driven algorithm usually has an arbitrary character: its columns are normally assigned to alphabet symbols according to some "lexicographic" ordering of the alphabet; while the rows are normally assigned to states more or less in the order in which these states are discovered when symbolically manipulating the automaton, or formulating the problem. Similarly, when implementing the equivalent hardcoded algorithm, while instructions relating to a given state are required to be grouped together, the order in which groups of such state-related instructions appear is as arbitrary as the assignment of rows to states in the table-driven case. This thesis is premised on the insight that such arbitrariness does not optimally use cache when the respective algorithms are run.

Thus, various alternative algorithms are proposed, that aim at better data organization in order to take advantage of the cache memory. In such context, the cache memory is maximally exploited in that data/instructions required for processing are present in the cache as much as possible, leading to the more rapid processing of the string. In order to achieve this, various implementation approaches are investigated whereby the cache principles of locality of reference are exploited[1]. The approaches yield various automata-based string processing algorithms such that some of them are proven to outperform their core counterparts when recognizing certain kinds of strings. Our work is thus based on the design and implementation of algorithms that attempt to account for likely cache utilization. Of course the attempt to account for cache utilization should be as efficient as possible so that the overheads caused during data/instructions organization is minimal compared to the overall algorithm's processing time. The performances of several of these algorithms are collected and cross compared with that of the conventional string recognition algorithms. The kind of automata used throughout this work is briefly discussed below.

1.2 The Kind of Automata

Throughout this thesis, we rely on Deterministic Finite Automata (DFA). Processing DFAs to determine string ownership is straightforward. Although string acceptance testing is also possible with non deterministic automata, we have deliberately chosen to restrict ourselves to DFAs. Therefore throughout this thesis, the various usage of the word Finite Automaton (FA) is in fact meant for Deterministic Finite Automaton.

1.3 Objective of the thesis

This work aims to exploit the principle of locality of reference on which optimal cache memory usage is based, in order to suggest alternative algorithms to the so-called conventional TD and HC algorithms. In order to achieve this, various implementation strategies that could be applied to TD and HC are suggested, resulting therefore in new algorithms. It is shown that, using a formal definition of a string recognizer to which implementation strategies are associated as variables, a *reservoir* of algorithms could be suggested. The suggested algorithms are then classified in a taxonomy tree based on a predefined relationship between them. The tree forms the basis for the design of an FA-based string processing toolkit that in the long run would be implemented and exploited for various computational needs. It is also shown that, some of the algorithms suggested outperform their core counterparts when input strings are made up of long sequences that has certain patterns. In the same context,

[1]Exploiting the principles of locality of reference makes it possible to code in cache at any given time to have a relatively high probability of referencing data in cache at that stage, and also, in determining the next instruction, control is likely to be passed to an instruction that is already in cache at that stage.

the performances of some of the suggested algorithms are cross-compared in order to establish the most efficient algorithm to date in relation to some input string pattern.

A long term objective of this work is to provide a complete implementation of the toolkit whose design is provided as part of this thesis. A domain specific language could be provided to allow the user to select and specify various operations available as part of the toolkit in order to perform tasks such as: benchmarking, performance and complexity analysis, application-specific usage, and the like. Since, the algorithms suggested are implemented taking into account the cache's locality of reference, it would be necessary to evaluate each algorithm on various platforms in order to determine the effect of caching in improving the algorithms. Another long term objective of our work would be investigations of the efficiency of the algorithms on various applications such as network intrusion detection systems, tandem repeat finders, DNA analyzers, spell checkers, etc. However, none of these themes are elaborated in this thesis. Instead, our work has been conducted by following the methodology described below.

1.4 Methodology

This thesis has its roots in a prior study that compared TD and HC [Nga03]. These two were regarded as core algorithms for the purposes of this study.

Initial investigations lead us to propose an alternative core algorithm that relied on both TD and HC —referred to below as the mixed-mode (MM) algorithm. As will be seen later, the algorithm is implemented in such a way that the whole transition set is subdivided in two disjoints subsets such that one is hardcoded and the other is table-driven. Then followed the provision of a DFA-based string recognizer's denotational semantics in terms of a mathematical function. To describe any one of the core algorithms (TD, HC or MM) certain variables of the function need to be appropriately chosen.

The next step, relied on previous investigations [Nga03, NWK03b, NWK03a] that revealed that the so-called core TD (resp. HC) algorithm is *memory-load* (resp. *instruction size*)[2] dependant, and thus, memory load (resp. instruction size) is a performance bottleneck. Furthermore, the correlation between HC and TD performance and hardware's cache [NKW06a] suggested that there is room for providing new algorithms that could exploit the principles of spatial and temporal locality of reference on which cache memory relies for fast processing of algorithms in general.

Therefore, an important part of the thesis is devoted to investigations of the various implementation strategies that are aimed at better data/instructions organization, with a view to improving the performance of an algorithm that is examining a string for acceptance testing. Associated with each implementation strategy, is a variable that is integrated into the original formalism for expressing a recognizer's

[2]By *memory-load* (resp. *instruction size*) dependant, we refer to the time taken by the processor to fetch data (resp. instruction) from memory. Since as the number of data/instructions available in memory increases, the time taken to fetch and execute them increases as well, resulting therefore to performance bottlenecks.

denotational semantics. Then follows provision of a unified formalism to define a recognizer's denotational semantics. It takes into account all the investigated strategies, associating one or more variables with each one of these strategies.

Such a unified formalism forms the basis for the suggestion of the various instantiated formalisms obtained by assigning values to the parameters of the unified mathematical function. Associated to each instantiated formalism is an FA-based string processing algorithm whose implementation is discussed. The suggested algorithms are then classified in the form of a taxonomy tree, which is further mapped into a toolkit whose architectural view is proposed in the thesis.

The foregoing matters are discussed in the first part of the thesis. The second part of the thesis is that of an empirical study of some of the algorithms suggested. It is in fact shown that some of the proposed algorithms outperform their associated core counterparts when processing large strings that follow a certain pattern.

The consequence of this work is the provision of a number of new algorithms, classified in a taxonomy, which forms the basis for a toolkit architecture. The complexity, performance and further refinement of each algorithm in the toolkit is a potential candidate for further intensive study in the field of DFA-based string recognition. Furthermore, the algorithms are also potential candidates for further studies in more specific application domains such as that of network intrusion detection, natural and computer virus scanning, tandem repeat finders, DNA analyzers etc.

1.5 Thesis Outline

The remaining part of this thesis is organized as follows:

- Chapter 2 of this current part I discusses some introductory formal elements of string processing that lead to a formal definition of string recognizers. Also provided in the chapter is a description of the operation of modern processors as well as a discussion of their performance metrics. These are needed to understand the cache spatial and temporal locality principle exploited in the design of the algorithms suggested throughout the thesis.

- In Chapter 3 of part II, the traditional string processing algorithms are revisited, namely the HC and TD algorithms. The algorithms are then used to suggest a new algorithm referred to as the mixed-mode (MM) algorithm. Then follows provision of the denotational semantics of all the core algorithms using a unified formalism from which any of the so-called core algorithms could be obtained.

- Chapter 4 introduces the first implementation strategy for implementing DFA-based string recognition. It is referred to as the dynamic state allocation strategy in which its associated table-driven, hardcoded and mixed-mode versions are suggested. Also suggested in this chapter is a new denotational semantics of the recognizer taking into account the DSA strategy as parameter variable, and it is shown that the core algorithms could be obtained.

- In Chapter 5, yet another implementation strategy is suggested, whose table-driven, hardcoded, and mixed-mode variations are discussed. As for the DSA strategy, it is shown that a denotational semantics could be provided whereby core algorithms are obtained.

- Chapter 6 concludes with the last implementation strategy referred to as the AVC algorithm following the same approach as that of the strategies discussed in the previous two chapters.

- In Chapter 7, all the strategies are combined together in order to produce a unified formalism for representing DFA-based string recognizers. It is shown that the formalism could be split into three independent formalisms representing all the TD, HC and MM algorithms respectively. Based on the defined formalisms, various new algorithms are suggested through instantiations of the parameter variables that form part of the unified formalism. The suggested algorithms are further compounded together and classified in a taxonomy tree that forms the basis for the design of a toolkit for FA-based string recognizers.

- Chapter 8 uses the constructed taxonomy tree from the previous chapter in order to map each node of the graph into a concrete class diagram, resulting therefore in a general class diagram representing an architectural view of the toolkit for FA-based string recognizers. Also discussed in this chapter are class components such as data members and their associated operations.

- In Part III of this thesis, we briefly discuss the empirically determined performance of some of the suggested algorithms. Although not all of the algorithms were implemented (due to the large number of possible algorithms), the foundations for setting up experiments used for comparing the algorithms is given in Chapter 9. Then follows in Chapter 10, experimental results where various graphs that depict the performance of selected algorithms compared to their core counterparts are discussed.

- The thesis is summarized in Part IV whereby a conclusion is provided in Chapter 11, and further directions for this work are discussed in Chapter 12.

CHAPTER 2

PRELIMINARIES

2.1 Introduction

The need to lay down foundations leading to a formal characterization of string recognizers[1] as well as the need to investigate performance bottlenecks of such recognizers is the key motivation of this chapter. Elementary notions intensively covered in the literature are revisited, introducing both a formal characterization of acceptors and the performance metrics that need to be considered for their efficient implementation.

Section 2.2 below deals with basic definitions necessary for a formal characterization of finite automata and therefore acceptors. Some definitions related to the performance of acceptor are also provided. Section 2.3, commences by covering terminology relating to computer organization, design and architecture that will be useful in understanding the performance of a string recognizer. Since most processors nowadays are superscalar, we provide a simple operational diagram that forms the basis for understanding architectural factors that may hamper the efficiency of acceptors. Brief discussions on how the identified components can affect the latency of string recognizers are provided towards the end of the section.

2.2 Formal elements of string processing

In this section, we provide basic elements of automata theory that are relevant in understanding the remainder of this work. String processing being the topic of concern, we do not attempt to cover the full extent of automata theory. Instead, we restrict ourselves to right linear languages and grammars that are the core foundation for modelling the solution to a *string processing* problem using finite automata.

Furthermore, string processing is a wide-ranging topic and is covered by many sources in the literature; often in relation to some area of specialization. While [Gus97, NR02, Wat95a] are sources that deal with string processing in computational biology, [Bac99, Cro02, FCH02, NN02] cover aspects of string processing in network intrusion detection systems. Sources such as [ASU86, AU73, WC93] depict aspects of string processing related to compiler construction and more precisely the lexical

[1]Throughout the thesis, the terms *string processor*, *acceptor* and *string recognizer* are use interchangeably to mean a finite automaton-based algorithm to test whether a word or string is part of a language modelled by a finite automaton.

analysis. Problems such as that of computational linguistics and spell checking in text are covered in [Hau01], while a more general coverage of the problem through the study of various algorithmic solutions and complexities of pattern matchers can be found in [CH97, NR02]. Many other problems that use string processing, such as natural and computer virus scanning, virtual circuit simulation and the like have also been covered in the literature, such as in [DC04, LMK+03] and [iCS03, MLLP03], respectively. The core foundation of formalisms related to automata theory in general has been extensively covered in sources such as [McN82, LP81, HMU01].

Below, the basic elements that relate to string processing at the algorithmic level are introduced in an ordered fashion. In order to do this, we start with simple concepts from discrete mathematic and computability theory leading to a more complete definition of an acceptor —which is the prime concern of this work.

Definition 2.1 (Set). A set is a collection of zero or more distinct objects, usually having some common characteristics of interest. The objects are generally referred to as elements. Unless explicitly stated otherwise, all sets considered in this thesis are finite —i.e. they have a finite number of elements. □

Definition 2.2 (Subset). Set $\mathcal{A}$ is a subset of set $\mathcal{B}$ ($\mathcal{A} \subseteq \mathcal{B}$), if and only if all elements of $\mathcal{A}$ are also elements of $\mathcal{B}$. □

Definition 2.3 (Empty set). A set with no element is said to be *empty* and is denoted by the symbol $\varnothing$. □

Definition 2.4 (Power set). Given a set $\mathcal{B}$, the power set of $\mathcal{B}$, denoted by $\mathcal{P}(\mathcal{B})$, is the set of all subsets of $\mathcal{B}$. Thus, $\mathcal{P}(\mathcal{B}) = \{\mathcal{A} : \mathcal{A} \subseteq \mathcal{B}\}$. □

Definition 2.5 (Cross-product). The cross-product of two sets $\mathcal{A}$ and $\mathcal{B}$, denoted by $\mathcal{A} \times \mathcal{B}$, is the set of all possible pairs that can be formed between elements of $\mathcal{A}$ and $\mathcal{B}$. Thus $\mathcal{A} \times \mathcal{B} = \{(a, b) : a \in \mathcal{A}, b \in \mathcal{B}\}$. □

Note that the cross-product operation is not commutative —i.e. $\mathcal{A} \times \mathcal{B} \neq \mathcal{B} \times \mathcal{A}$. Furthermore the cross-product of three sets is a set of triples, and inductively, the cross-product of n sets is a set of n-tuples.

Definition 2.6 (Alphabet). An alphabet is a finite non-empty set of symbols. □

Definition 2.7 (String or word). Given an alphabet $\mathcal{V}$, a string over $\mathcal{V}$ is a finite sequence of elements of $\mathcal{V}$. □

Definition 2.8 (Length of a string). The length of a string w, denoted by $|w|$, is the number of symbols (in the string). □

Definition 2.9 (Empty string). An empty string denoted by ϵ is a string of length zero. □

Definition 2.10 (Set of all strings). Given an alphabet $\mathcal{V}$, we define $\mathcal{V}^*$ (respectively $\mathcal{V}^+$) to be the set of all strings over $\mathcal{V}$ including the empty string (respectively excluding the empty string). □

Definition 2.11 (Language). Given an alphabet $\mathcal{V}$, any subset $\mathcal{L}$ of $\mathcal{V}^*$ is a language over $\mathcal{V}$. □

Definition 2.12 (Kleene star and Plus). The Kleene star (respectively Kleene plus) is a unary operation, either on sets of strings or on an alphabet. The application of the Kleene star (respectively Kleene plus) to a set $\mathcal{V}$ is written as $\mathcal{V}^*$ (respectively $\mathcal{V}^+$). If $\mathcal{V}$ is a set of strings then $\mathcal{V}^*$ (respectively $\mathcal{V}^+$) is defined as the smallest superset of $\mathcal{V}$ that contains zero or more concatenated elements of $\mathcal{V}$ (respectively one or more concatenated elements of $\mathcal{V}$). □

Remark 2.13 (Regular Expression). A regular expression is a formal way of specifying a regular language using an operator such as Kleene star ($*$), Kleene plus ($+$), Union, Intersection, Complementation, Difference, Reversal and Power [Wat95b, HMU01, WC93]. □

Definition 2.14 (Grammar). A grammar $\mathcal{G}$ is a 4-tuple ($\mathcal{N}$,$\mathcal{V}$,$\mathcal{R}$,S) where:

- $\mathcal{N}$ is a finite set of symbols called the non-terminals or variables;
- $\mathcal{V}$ is a set of alphabet symbols —referred to as terminals or constants— which are disjoint from $\mathcal{N}$ (i.e. $\mathcal{N} \cap \mathcal{V} = \varnothing$);
- $\mathcal{R}$ is a finite set of rules of the form $lhs \rightarrow rhs$, where $lhs \in (\mathcal{N} \cup \mathcal{V})^+$, lhs contains at least one symbol in $\mathcal{N}$, and $rhs \in (\mathcal{N} \cup \mathcal{V})^*$;
- S is the start non-terminal or variable. □

Definition 2.15 (Derivation). A derivation is a string of terminal symbols that can be derived through a succession of rule-based substitutions, starting with the start symbol of a grammar. In each case, a substring of the string derived to date that matches the *lhs* of some rule is replaced by the *rhs* of a rule. □

Definition 2.16 (Language generated by a Grammar). A language generated by a grammar is the set of all possible derivations. □

Definition 2.17 (Right Linear Grammar). A grammar is right-linear if each rule is of the form: $A \rightarrow wB$ where A is a non-terminal, w is a string of terminals, and B is a string with at most one non-terminal. □

Definition 2.18 (Right Linear Language). A right linear language is a language generated by a right linear grammar. □

Definition 2.19 (Finite Automaton). A Finite Automaton (FA) is a 5-tuple ($\mathcal{Q}$,$\mathcal{V}$,Δ,s_0,$\mathcal{F}$) where:

- $\mathcal{Q}$ is a finite set of states;
- $\mathcal{V}$ is the set of alphabet symbols;
- Δ is a transition function, as discussed below;

- $s_0 \in \mathcal{Q}$ is the start state;
- $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states. □

Δ corresponds to the function[2], $\delta : \mathcal{Q} \times \mathcal{V} \nrightarrow \mathcal{Q}$. Thus, strictly speaking, the elements of Δ are *pairs* of the form $((q_i, c), q_j)$, where $q_i \in \mathcal{Q}$, $q_j \in \mathcal{Q}$ and $c \in \mathcal{V}$. However, for ease of reference, these elements are generally flattened to form triples: (q_i, c, q_j). In the remainder of this thesis, Δ should be seen as referring to a set of such a triples, while $\delta(q, c)$ represents the unique third element of a matching triple in Δ.

As is well-known, an FA is said to *accept* an input string formed from elements of $\mathcal{V}$, say $c_0.c_1 \cdots c_{n-1}$, if successive applications of the transition function to elements of the string result in a final state (i.e. if $\delta(\cdots(\delta(\delta(s_0, c_0), c_1)\cdots), c_{n-1}) \in \mathcal{F}$). An input string that is not accepted is said to be *rejected*.

In general, rejection of an input string can occur for one of two reasons: either the sequence of transition function applications does not lead to a final state; or somewhere along that sequence a state and input symbol combination, (q, c_i), is reached for which δ is not defined.

For the purposes of this thesis, a special "sink" state, say σ, will be assumed such that all transitions into this state imply rejection[3], while all transitions terminating in any other state imply acceptance. Thus, we assume that $\delta : \mathcal{Q} - \{\sigma\} \times \mathcal{V} \rightarrow \mathcal{Q}$ is a total function, and that $\mathcal{Q} - \{\sigma\} = \mathcal{F}$. These assumptions were made to facilitate simulations in which the prime concern was about generating FA's for experimental purposes to examine performance issues, rather than about niceties of how final states are reached in various applications.

Theorem 2.20 (Kleene's Theorem). Every regular language can be recognized by a finite automaton. Every finite automaton recognizes a regular language. □

Definition 2.21 (Acceptor). An acceptor (or a string recognizer) of a finite automaton is an algorithm that relies on the finite automaton's transition function in order to determine whether a string is part of the language modelled by the FA or not. □

An acceptor of the automaton $\mathrm{M} = (\mathcal{Q}, \mathcal{V}, \Delta, s_0, \mathcal{F})$, where $\mathcal{L}(\mathrm{M}) \subseteq \mathcal{V}^*$ is the language of M, can be characterized by the following function:

$$\rho : \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q}) \times \mathcal{V}^* \rightarrow \mathbb{B} \tag{2.22}$$

$$\rho(\Delta, s) = \begin{cases} true & \text{if } s \in \mathcal{L}(\mathrm{M}) \\ false & \text{if } s \notin \mathcal{L}(\mathrm{M}) \end{cases}$$

where $\mathbb{B} = \{true, false\}$ is the set of boolean values. In fact, ρ is the *denotation semantics* of the acceptor [Mey90].

[2] In this thesis, the symbol $\rightarrow$ is used in the signature of a total function whereas $\nrightarrow$ is used for a (possibly) partial function. We explain later why it will be convenient to assume a total function in this thesis, even though the general FA definition does not have this as a requirement.

[3] In later experiments, this state will typically be designated by the integer -1.

The denotational semantics indicates the "meaning" of the algorithm in functional terms, but hides details about how the algorithm that performs acceptance testing should actually work. At this level of description, the acceptor is viewed as a "black box" that receives as input a transition set and a string, and later produces a boolean as output. Details are hidden about a processing phase in which each symbol of the string is scanned in order to establish subsequent states of the automaton.

There are, in fact, a large number of ways in which this processing can take place, as will be discussed in later chapters. Two approaches will be followed to inform the reader about various ways of carrying out this processing. At the more concrete level, an algorithmic description will be given. However, as a complementary approach, the general denotational semantic description given in (Equation 2.22) above will be refined to be more specific to its associated algorithm. Each refinement will replace ρ with a function that has additional arguments in its domain, and each argument added will provide more information about how the associated algorithm is to be carried out.

Definition 2.23 (Complexity of acceptors). The time complexity of an acceptor depends on the length of the string being tested for acceptance. The worst case complexity is bound from below by $O(m)$, where m is the length of the string. □

It is fairly easy to design a string recognizer whose complexity —in theory— is $O(m)$. In practice, however, one may discover that it does not necessarily comply with this theoretical complexity. In effect, the performance of an algorithm largely depends on various factors such as hardware resources and, more importantly, on the way in which the algorithm has been implemented. Such factors may retard its overall performance in a non-linear fashion. In this thesis, we shall argue that a maximally efficient string recognizer should not only be based on traditional good software and algorithmic design considerations, but also on a thorough knowledge of the capabilities offered by the hardware on which the recognizer will be processed.

Remark 2.24 (Performance of an acceptor). The performance of a string recognizer on a given hardware platform is indicated by the number of *clock cycles* (ccs) required by the recognizer to perform some task. □

Given the performance of some acceptor, we can easily estimate the corresponding processing time in seconds, if there is knowledge about the processor's capacity in megahertz. For example, a program that requires 100 clock cycles to complete under a 100MHz processor has consumed about 1μ seconds, representing the processing time.

The operation on the widely used type of processors nowadays as well as their performance metrics is depicted in the next section.

2.3 Operation of superscalar processors and performance metrics

This section provides the operational diagram of the most popular type of processor in use nowadays, referred to as a superscalar processor. Using this diagram,

various aspects of the hardware are investigated that directly participate in the execution life cycle of acceptors and hence relate to performance metrics. In effect, in order to understand factors affecting the efficiency of string recognizers, and subsequently provide some performance improvements solutions, one needs to understand where the processing time is spent in the hardware. In certain circumstances, the time wasted largely depends on the hardware. Therefore hardware design solutions can be provided in order to speed up the recognizer. However, some performance bottlenecks can also be overcome at software level —i.e. the FA implementer may be able to use some algorithmic and fine tuning strategies for performance enhancement.

More specifically, although there is nothing acceptor implementers can do at hardware level, some factors such as caching, pipelining and branch prediction can potentially be exploited at the software level in order to improve performance, hence the importance for investigating performance metrics at both hardware and software levels.

In the subsection below, we briefly introduce the basic principles of computer architecture that relate to potential algorithmic solutions to a specific problem and its applicability on hardware.

2.3.1 Basic principles of computer architecture

We briefly present basic concepts of computer architecture that are relevant in this thesis. A more advanced and detailed coverage on the topic may be found in [DWM00, PH05, HP03, HVZ02].

The understanding of the functioning of hardware components is an important factor for program performance improvement, not only for the identification of hotspots in a given program but also for the better utilization of the various hardware resources that participate in the program execution process. Computer architecture which may be regarded as the logical principle underlying the overall design of a computer in order to enable efficient and effective interaction of hardware components is usually considered as those attributes of the hardware that have direct impact to the logical execution of a program.

In order to have a program executed by a computer, the programmer is supplied with the hardware's Instruction Set architecture (ISA), which includes among others: active datatypes, instructions, registers, addressing modes, memory architecture, interrupts and exception handling, and external I/O (if any). An ISA is a specification of the set of all binary codes (opcodes) that are the native form of commands implemented by a particular CPU design. The set of opcodes for a particular ISA is also known as the machine language for the ISA. An ISA can also be emulated on another computer by an interpreter. Due to the additional translation needed for the emulation, the process is usually slower than directly running programs on the hardware implementing that ISA. Today, it is common practice for vendors of new ISAs or microarchitectures to make software emulators available to software developers before the hardware implementation is ready.

During program execution, the opcodes (Operations Code) operate on registers, values in memory, values stored on the stack, I/O ports, etc. They are used to perform

arithmetic operations and move and change values. Operands are the things that opcodes operate on. Microprocessors perform operations using binary bits. When the opcode values are active at the decoder's logic inputs, the desired operations are performed. Each of these operations is assigned a numeric code, which is the opcode. To assist in the use of these numeric codes, mnemonics are used as textual abbreviations. It is much easier to remember the mnemonic ADD than the corresponding opcode 05, for example.

For a program to be executed by a computer, it is supplied with a system of codes directly understandable by a computer's CPU, also referred to as the CPU's native language or *Machine Language*. The "words" of a machine language are called instructions. Each of these causes an elementary action by the CPU, such as reading from a memory location. A program is just a long list of instructions that are executed by a CPU. Older processors executed instructions one after the other (scalar processors), but newer processors are capable of executing several instructions at once (superscalar processor).

At the execution level, opcodes are decomposed into microinstructions (or microcode instructions) which are the most elementary computer operations that can take place —for example, move a bit from one register to another. It takes several microinstructions to carry out one machine instruction. A microinstruction asserts a set of control signals that are active on a given clock cycle. It also specifies what microinstruction to execute next. A set of micro-instructions that control a processor is referred to as Microcode.

Most of the superscalar processors nowadays rely on various sophisticated techniques used to enhance the performance of a running program. Various performance enhancement techniques are described in detail in [PH05, HP03, Ger98]. The following two technics are among those[4]:

- **Pipelining:** This is an implementation technique in which multiple instructions are overlapped in execution, much like in an assembly line. An instruction pipeline is a technology used on microprocessors to enhance their performance. Pipelining greatly improves throughput. It ensures that the processor never needs to wait for instructions to be fetched and decoded before being executed; as soon as an instruction is executed, another is waiting.

- **Branch prediction**: In computer architecture, a branch predictor is the part of a processor that determines whether a conditional branch in the instruction flow of a program is likely to be taken or not. Of course although each branch of a branching instruction has some likelihood (probability) of being taken, there is a variability between the probability associated with each branch. At a point in time, a branch associated with the highest probability would be considered as the one to be taken, provided that the prediction corresponds to the logical

[4]It should be noted that many of the architectural components described here have been taken from Intel literature (http://developer.intel.com/design/Pentium4/documentation.htm), and often from literature that relates to the pentium 4. There will no doubt be various difference from one vendor to the next. Nevertheless, the overall architectural concepts are similar.

flow of the program. The target of each branching instruction associated with its probability is kept in the Branch Target Buffer (BTB). Branch predictors are crucial in today's modern, superscalar processors for achieving high performance. They allow processors to fetch and execute instructions without waiting for a branch to be resolved. The processor fetches branches that are likely to be taken from the Branch Target Buffer (BTB). In some circumstances the target to be taken may be the wrong one, this is referred to as a *mis-prediction*, which is resolved by taking the correct branch as indicated in the program being executed. A mis-prediction is always associated with some latency penalty.

One of the crucial components for performance enhancement at hardware level is the cache. To capitalise on its tight integration with the processor, it holds frequently accessed data and frequently executed instructions thereby minimizing the time spent for data/instructions fetches during instruction execution. For performance enhancement, the cache relies on the principles of spatial and temporal locality of reference. The temporal locality of reference is based on the premise that data/instruction currently being processed is likely to be processed again in the near future. For the spatial locality of reference or locality in space, data/instructions are fetched from memory in chunks, so that not only the data/instruction immediately sought by the processor is copied into cache, but also other data and instructions that are stored in physical proximity. This mechanism assumes that data/intructions closer to the one currently being processed are likely to be processed in the near future.

At the software level, caching can sometimes be exploited to improve a program's performance. Such exploitation would require the organization of data/instruction such that both temporal an spacial locality of references are accounted for, at processing time.

Beside a processor's conventional cache memory, many other components in the hardware's execution path behave as a sort of cache. (See [PH05, HP03] for details of those components.) The most prominent ones are:

- **The Trace Cache Buffer (TCB)** which is an "instruction cache" that holds a sequence of instructions with a given starting address, and a sequence of branch outcomes which describe the path followed by the programm's instruction stream [RBS99]. In some architecture such as the recent pentium, the trace cache buffer holds microoperations rather than opcodes;

- **The Translation Lookaside Buffer (TLB)** This is a special unit that translates virtual addresses into physical ones that have to be used for physical memory access. The buffer also acts as a cache in the sense that, for performance enhancement, it holds the most recently used addresses so that these can be accessed without further translation.

2.3.2 Operation of a superscalar processor

As opposed to a scalar processor, superscalar processors are designed both to execute a set of micro-operations within a single processor's clock cycle (using pipelining

at the hardware level), as well as to simultaneously dispatch micro-operations to appropriate units of the processor. Superscalar processors therefore have built in execution units such as integer, floating-point, load and store units. These unit are separated from each other not only to modularize the execution logic but also so that several instructions can be executed at a time.

The overall operation of superscalar processor can be summarized as follows:

1. The processor fetches instructions in the form of micro-operations from the TCB. The instructions usually come from the trace predictor unit (TPU) —a unit that holds the predicted instruction traces coming either from the branch target buffer or any other instruction trace that does not depend on branching.

2. Each block of micro-operations is then transferred to its appropriate execution unit for processing.

3. The results are written back to memory according to the order of execution of the program.

2.3.3 Operational diagram of a superscalar processor

Figure 2.1 depicts a more detailed operational flowchart of a superscalar processor. The proposed operational diagram relied on the so-called traditional fetched-execute diagram in [PH05, HP03]. It has been modified and expanded here to depict explicitly the participation of some of the hardware's components that could be considered as "time consumers" during program execution. Of course, hardware aspects such as pipelining, branch prediction, bus structure, and the like are not depicted for consistency, although they are also regarded as critical when it comes to evaluating program's latency. In our diagram, The TCB is the principal source of instructions (micro-ops) that feed the processor's execution units before pipelining takes place. In general, micro-codes come from the trace predictor unit that produces trace identifier containing the starting address of the program counter (PC), as well as all conditional branches embedded in the trace. According to the starting PC and the data throughput between the TCB and TPU, multiple traces can be feed into the TCB at a time. In the same way, several micro-ops can be fetched at a time from the TCB for execution in a pipelined fashion.

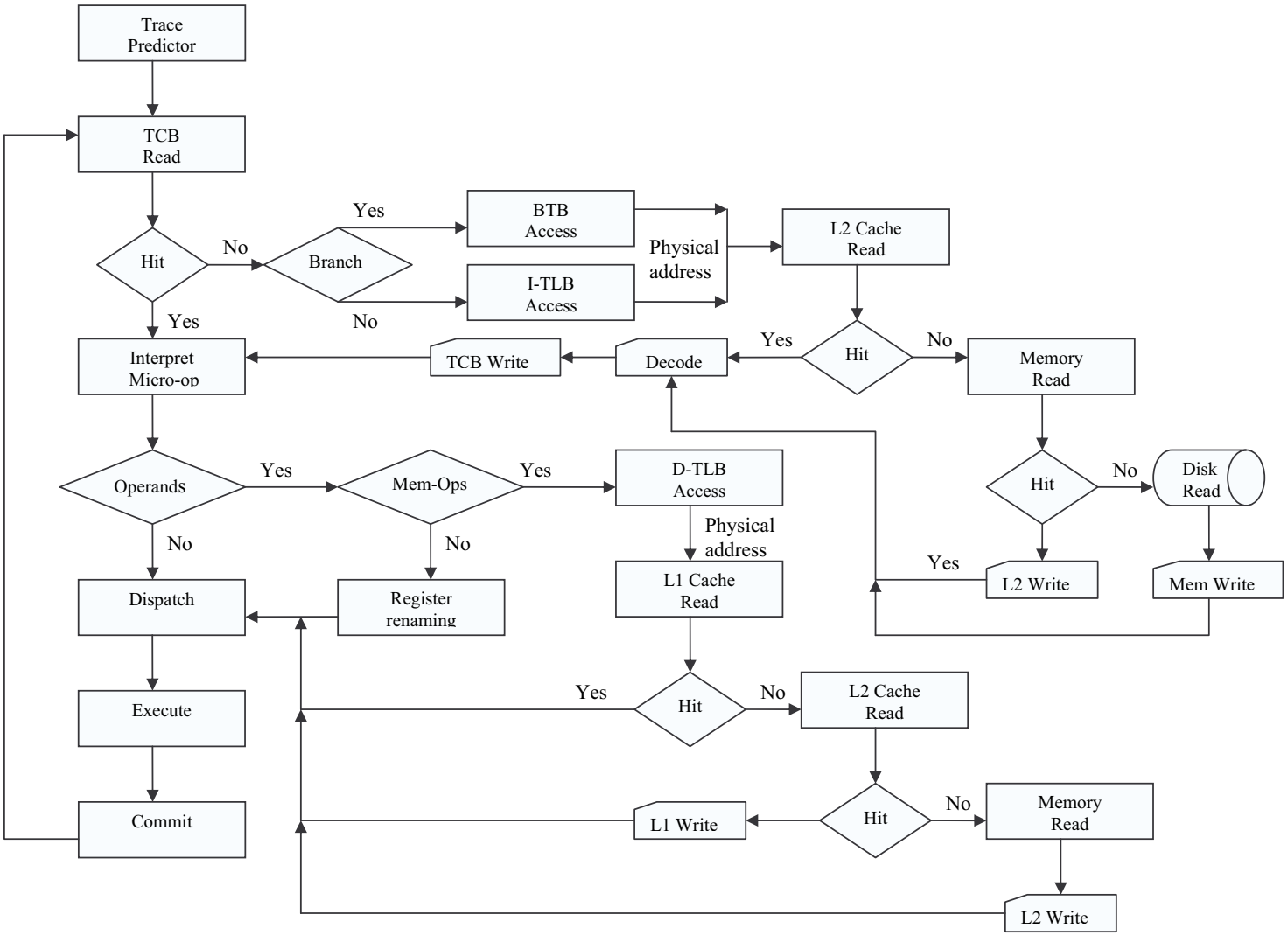

Figure 2.1. Operation diagram of superscalar processors based on information gathered from [PH05, HP03]

As shown in the diagram, the performance of a program depends on the path followed by the processor at run-time before executing a given instruction. The best scenario would be the case where the instruction being sought is found in the trace cache buffer, and its execution somehow does not require an operand. The instruction is then executed in the appropriate unit and response is send back to the control unit for appropriate action to be taken.

However, it should be noted that a trace being sought from the TCB is not always available. In this case, a miss will occur (and therefore a time penalty incurred), requiring the instruction trace to be fetched from other units. The process may be even worse if the desired trace is only available in the hard drive of the computer. It follows that the latency of a running program is a function of the complexity of the processing path to follow at run-time. The subsection below extract from the diagram various hardware performance metrics that should be given attention when evaluating the overall performance of a given program.

2.3.4 Performance metrics identification

Using the operational diagram described in the previous section, it is easy to identify where time is spent during the execution of a given program. The speed at which the execution unit processes the fetched micro-ops from the TCB depends on the kind of the instructions to be executed. The following scenarios can therefore be envisaged during the processing of an instruction:

- If the instruction is so simple that it does not require operands, the instruction is directly transferred to its appropriate execution unit for processing. This corresponds to a flow down the left hand column of the diagram —i.e. from trace predictor, to TCB read, to interpret micro-op, to dispatch, execute and commit.

- In general, an instruction may require register operands or references to memory addresses (for data manipulation). In case the operands are all CPU-registers, the processor need only rename the registers and map them to their corresponding hardware reference. (This corresponds to the No-branch from the Mem-ops choice box in the diagram.) However, if the operand references to data originating in memory, a complete search process will start (corresponding to the Yes-branch from the Mem-ops choice box in the diagram), involving the following:

  - *Data-TLB:* The data translation lookaside buffer is accessed in order to convert the virtual address of the datum being sought into a physical address used to access the L1 cache. In case that virtual address is not found in the D-TLB, reference is made directly to the L1 cache for data retrieval. It then follows an update of the D-TLB in case of a successful read into L1.

  - *L1 Data cache:* The data being sought may be found into the L1 data cache. Otherwise, a miss penalty occurs and the data is now sought from the L2.
  - *L2 Data cache:* The data may be found in the L2 cache. Otherwise the data has to be fetched from main memory.
  - *Main memory:* The sought data may be found in memory, otherwise a search has to be made on the disk.
  - *Disk:* This is the lowest level of memory in the hierarchy and it is the slowest one. When the data being sought is found on a lower level of the memory hierarchy, the memory at the higher level in the hierarchy is updated for further usage.

- If the instruction is not found in the TCB (i.e. the no-branch of the Hit choice box below the "TCB read" box in the diagram is taken), a different scenario for moving data between CPU and data-cache arises, this time between CPU and Instruction cache. The first unit to be accessed in this case is either the I-TLB or the BTB. For a normal (non-branching) instruction the virtual address is converted by the I-TLB. The BTB is accessed when the instruction appears to be a branch instruction. Both units reference memory (hierarchically) for instruction retrieval, firstly trying the L2 cache, then the internal memory, and then the disk. Once the sought instruction is found, the decoder then converts the instruction into micro-operations. Then follows an update of the TCB for further usage.

The operational information in Figure 2.1 suggests that the neat abstract analysis of algorithmic performance could be significantly distorted by the detailed way in which these lower levels of hardware are actually deployed during processing. These matters could affect not only the constants in the order-of-magnitude theoretical estimates, but even the very order-of-magnitude estimates themselves. For example, high cache miss rates could change a theoretically linear algorithm into one that exhibits super-linear performance in practice. Even the performance of simple algorithms such as string recognizers may be affected, (as will be later reported in this thesis) due to the size of the underlying automaton and/or the length of the string to be analysed.

There are a range of performance metrics that could be relevant in trying to understanding the performance of various applications. These include:

1. Miss rate in the TCB
2. Misprediction rate in the BTB
3. Access rate in the I-TLB
4. Access rate in the D-TLB
5. L1 data cache Miss rate
6. Instruction miss rate in L2 cache

7. Data miss rate in L2 cache

8. Data, instructions and branches hazard

Ideally, in order to optimize the performance of a given application, the designer should take account of all the above metrics when designing an implementation. In this thesis, various implementation strategies for finite automata-based acceptors are proposed, with a view to exploiting the capabilities offered by superscalar processors. We rely on the notion of *better data/instructions organization in cache* in order somehow positively affect some of the performance metrics discussed in this chapter through various implementation strategies of string recognizers. As a result of this approach, a variety of algorithms is provided and constitutes a potential source for the study of the effects of the various performance metrics identified in this chapter– and could subsequently yield to recommendations in regard to the kind of hardware resources that could improve string recognition performance.

As a disclaimer to this part of the thesis, it is recognized that a fuller treatment of computer architecture as well as the effect of hardware components on the latency of a program should include a discussion on matters such as: instruction set size/format; data/instruction movement between CPU and main memory, and between CPU and cache; the kind of cache being utilized at hardware level (unified or split cache); the replacement algorithms on which the cache is based upon; the mapping principle on which the cache relies (direct mapping, associative mapping, etc); the instruction pipelining strategies; and finally the effect of branch prediction on a running program. However, these matters are beyond the scope of the present thesis. Again, more information on those issues can be found in sources such as [PH05], [Hyd03], and of course the various online Intel reference manuals at www.intel.com.

2.4 Summary of the chapter

In this chapter, we have used several basic concepts in the literature to provide the denotational semantics of string recognizers. We have also depicted the overall operational diagram of superscalar computer used nowadays. The diagram suggested various performance metrics that could be considered when designing acceptor implementations. In later chapters, we exploit the effects of cache optimization with the aim of providing fine tuning strategies for enhancing the performance of FA related problems. In the next chapter, various FA-based string recognition algorithms are presented, each based on specific implementation strategies. Of course, the already-provided function to represent the denotational semantics of a generic string recognizer remains a legitimate denotational description of each of these alternative algorithms. However, in each of the forthcoming chapters, a more explicit functional description of each of the algorithms will be given as an alternative, less abstract, denotational description of the algorithm. These functional descriptions will eventually serve as the basis for a taxonomy of FA-based string recognizers.