Optimization Methods on Embedded Systems

Ph.D. dissertation

by

Ferenc Havasi

Supervisor: **Prof. Tibor Gyimóthy**

Submitted to the

Ph.D. School in Computer Science

Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged, 2013.

Preface

About the thesis

In life, I have always tried to create really useful things. At the Software Engineering Department, this led me to work on embedded systems, which I have been investigating for over a decade.

In the world of embedded systems the resources are usually quite limited, so in many areas optimization is very important. Really effective optimization cannot be achieved without scientific research. It gave me the motivation to achieve the results presented here and try to make it work in practice as well.

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Tibor Gyimóthy for his professional and personal support, as without him this thesis would not have been possible. Secondly, my thanks goes to my colleagues Dávid Tengeri, Zoltán Sógor, Miklós Kálmán and Tamás Gergely. Lastly, I would like to express my gratitude to my parents János and Margit, my sister Mónika, my companion Barbara, and my coach Márta.

Ferenc Havasi, December 2013.

Contents

List of Tables

List of Figures

Introduction

Embedded systems are a combination of computer hardware and software, with a dedicated function within a larger mechanical or electrical system, often with hard performance constraints. These kinds of devices have more and more applications in today's world. Cell phones, DVD players, MP3 players, GPS receivers, car electronics and similar devices play an important role in our lives nowadays. A good indication of the penetration rate of embedded systems is that in 2002 only 2% of the microprocessor were used in personal computers, and the rest (98%) in embedded systems, and since then even the rate has increased [42]. The economic and social impact of this on societies will increase, and new results may have great practical significance.

One of the main features of these devices is that their resources are generally much more limited than the resources of personal computers. So optimizing the software components is always a priority. In this thesis, three key results are presented, namely one XML-related result and two flash file system optimizations.

	Thesis	**Publication**
1.	XML Semantic Extension and Compaction	[20], [26], [27]
2.	Size Optimization with ARM Code Compression	[15]
3.	Performance Optimization with an Improved B+ Tree	[19]

Table 1: Theses and publications

1. XML Semantic Extension and Compaction

XML is one of the most popular, general, and widely used document formats even on desktop computers and embedded systems. The number of applications capable of storing things in XML format is growing quite rapidly, and it has now become one of the de facto standards of structured text formats.

XML uses elements and attributes to structure text information. An XML attribute has a name, and a value. Before our solution, there was no way to define semantic rules for their computation. To bridge this gap, we defined a new metalanguage called SRML (Semantics Rule Meta Language), where semantic computation rules can be defined for XML attributes.

Using SRML it is not necessary to store the computable XML attributes, so XML documents can be stored in a more compact format. Using our method as a preprocessor of the XMill XML compressor, we can improve its efficiency by 9-26%, because our method can identify correlations that cannot be identified by the usual XML compressors.

We present our results in Chapter 1, which were published in [20], [26] and [27].

2. Size Optimization with ARM Code Compression

Some years ago embedded systems had very limited storage resources in general. Nowadays, because of the low cost of flash chips, the relative importance of the size factor has decreased in the case of multimedia devices, but in the case of the functional devices it is still high.

One of the solutions available for handling size problems is compression. Because one of the most popular embedded system architectures is ARM, we designed a new ARM code compressor algorithm. It is a model-based compressor that uses decision tree as a model, and its coder is an arithmetic coder.

It was implemented in the JFFS2 file system, with an effective combination structure with the original compressor (zlib). We achieved a 13-19% size reduction compared to the original compressor, but the boot time doubled or quadrupled relative to the original.

The implementation of the transparent compression framework is now a part of the JFFS2 and the Linux kernel. The ARM code compressor was patented with US Patent number 6,917,315. [40]

We will present our results in Chapter 2, which were published in [15].

3. Performance Optimization with an Improved B+ Tree

Most mobile devices handle data files and store them on their own storage device. In most cases, this storage device is flash memory. On smart devices an operating system helps to run programs that use some kind of file system to store data. The data capacity, the response time (how much time is needed to load a program) and the boot time (how much time is needed to load all the necessary code and data after power up) of the device depend on the properties of the file system. All of these parameters are important from the viewpoint of device usability.

The size of the flash chips in embedded systems have dramatically increased. This led to the earlier flash file systems solutions (such us JFFS2 file system) being practically unusable in the case of large flash chips (over 512MB) because of its slowness and RAM consumption. Hence it was necessary to design a new flash file system, and it required new algorithms and data structures to make its performance more optimal and provide a power loss safe solution.

Our new result is an improved B+ tree that was implemented in the UBIFS file system. It became a part of the Linux kernel, and it is used in the Nokia N900 smart phone.

We present our results in Chapter 3, which were published in [19].

Figure 1: The Nokia N900 smart phone

Chapter 1

XML Semantic Extension and Compaction

These days XML is one of the most popular, general, and widely used document formats even on desktop computers, servers and embedded systems, or in communication protocol among them. Applying a result we achieved here, XML files can be stored more efficiently, which can be very useful especially in the case of embedded systems or network applications.

We designed a new metalanguage called SRML (Semantic Rule Meta Language) that allows one to define a real XML attribute via semantic rules. These SRML rules describe how the value of an attribute can be calculated from the values of other attributes. They are quite similar to those of the semantic rules of Attribute Grammars [28], and can be used for compacting an XML document by removing computable attributes.

The generation of these SRML files can be performed manually (if the relationship between attributes is known) or via machine learning methods. The method looks for a relationship among the attributes and for patterns in them using specific rules.

During the testing of our implementation, the input XML files were compacted to 70-80% of their original size, while maintaining further compressibility (e.g. the XMill XML compressor was able to compress this file after first being compacted).

1.1 Preliminaries

Here, we will talk about XML files along with the necessary preliminaries for Attribute Grammars. Both will be needed to better understand what is discussed in later sections.

1.1.1 XML

The first concept that must be introduced is the XML format. A more thorough description of the XML documents can be found in [7] and [16]. XML documents are very similar to *html* files, as they are both text-based. The components in both are called *elements*, which may contain further *elements* and/or text, or they may be left empty. *Elements* may have attributes like the html anchor tag *a* attribute of *href* elements in html files. In *Figure 1.1*, there is an example for storing a numeric expression in XML format. This example has an additional attribute called *"type"*, which stores the type of the expression. The values can be *int* or *real*.

```
<expr> <multexpr op="mul" type="real">
 <expr type="int"><num type="int">3</num></expr>
  <expr type="real">
   <addexpr op="add" type="real">
    <expr type="real"><num type="real">2.5</num></expr>
    <expr type="int"><num type="int">4</num></expr>
   </addexpr>
  </expr>
</multexpr> </expr>
```

Figure 1.1: A possible XML form of the expression $3 * (2.5 + 4)$.

It is possible to define the syntactic form of XML files. This is achieved through a DTD (Document Type Definition) or XSD (XML Schema Definition) file.

During our research the DTD format was generally used, so we described our results using DTD.

A DTD file specifies the accepted format of the XML document. If an XML document uses a DTD, all elements and attributes in the XML document must conform to the syntactic validity of the DTD. The DTD of *Figure 1.1* is listed in *Figure 1.2*.

```
<!ELEMENT num (#PCDATA) >
  <!ATTLIST num type ( real | int )#REQUIRED >
<!ELEMENT expr ( num | multextr | addexpr ) >
  <!ATTLIST expr type( real | int ) #IMPLIED >
<!ELEMENT multexpr ( expr , expr ) >
  <!ATTLIST multexpr op ( mul |div ) #REQUIRED
            type ( real | int ) #IMPLIED >
<!ELEMENT addexpr ( expr , expr ) >
  <!ATTLIST addexpr op ( add |sub ) #REQUIRED
            type ( real | int ) #IMPLIED >
```

Figure 1.2: The DTD of the simple expression in *Figure 1.1*

A DOM (Document Object Model) [43] tree is a tree containing all the tags and attributes of an XML document as leaves and nodes (*Figure 1.3* is the DOM

tree of *Figure 1.1*). This DOM tree is used by the XML processing library for internal data representation. DOM is a platform- and language-independent interface that allows the dynamic accessing and updating of the content and structure of XML documents. When DOM tree parsing is used, it makes the XML document handling easier, but it requires more memory to accomplish this, since it creates a tree of the XML in the memory. This method is quite effective on smaller XML documents.

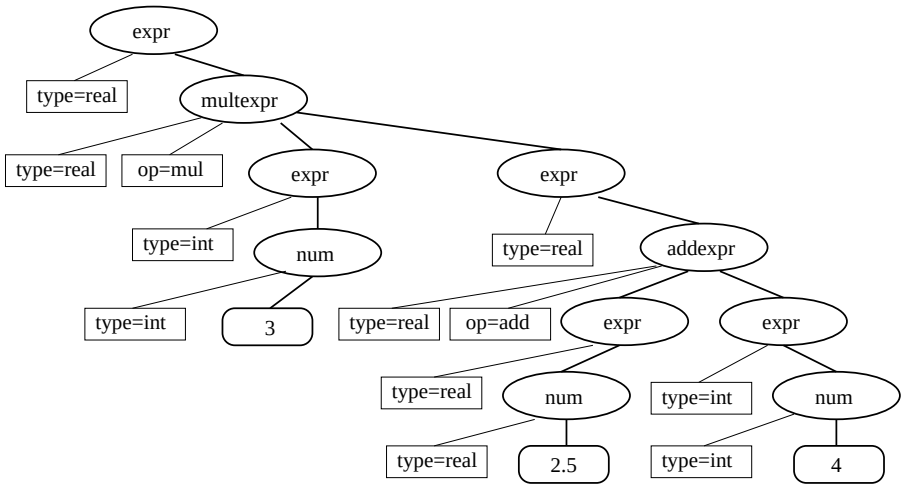


Figure 1.3: XML document DOM tree

1.1.2 Attribute Grammars

Another key concept that should be mentioned is that of Attribute Grammars. Attribute Grammars are based on the context-free grammars. Context Free (CF) Grammars can be used to specify derivation rules for structured documents. A *CF Grammar* is a four tuple $G = (N, T, S, P)$, where N is the set of nonterminal symbols, T is a set of terminal symbols, S is a start-symbol and P is a set of syntactic rules. It is required that on the left side of every rule only one nonterminal can be present. Given a grammar, a derivation tree can be generated based on a specific input. The grammar described below specifies the format of the simple numeric expression, shown in *Figure 1.1*.

```
N = { expr, multexpr, addexpr, num }
S = expr
T ={"ADD","MUL",INTNUM, REALNUM}
P :
 (1) expr     -> num
 (2) expr     -> multexpr
```

```
(3) expr     -> addexpr
(4) addexpr  -> expr "ADD" expr
(5) addexpr  -> expr "SUB" expr
(6) multexpr -> expr "MUL" expr
(7) multexpr -> expr "DIV" expr
(8) num      -> INTNUM
(9) num      -> REALNUM
```

An *Attribute Grammar* contains a CF grammar, attributes and semantic rules. The precise definition of Attribute Grammars can be found in [28] [2]. In this section, we will only mention those definitions which are required for a better understanding of later parts of the thesis.

An attribute grammar is a three tuple $AG = (G, AD, R)$, where

1. $G = (N, T, S, P)$ is the underlying context-free grammar.

2. $AD = (Attr, Inh, Syn)$ is a description of attributes. Each grammar symbol $X \in N \bigcup T$ has a set of attributes $Attr(X)$, where $Attr(X)$ can be partitioned into two disjoint subsets denoted by $Inh(X)$ and $Syn(X)$. $Inh(X)$ and $Syn(X)$ denote the inherited and synthesized attributes of X, respectively. We will denote the attribute a of the grammar symbol X by $X.a$.

3. R orders a set of evaluation rules (called *semantic rules*) to each production as follows: Let p: $X_{p,0} \ldots X_{p,n_p}$ be an arbitary production of P. An attribute occurrence $X_{p,k}.a$ is said to be a *defined occurrence* if $a \in Syn(X_{p,k})$ and k=0, or $a \in Inh(X_{p,k})$ and $k > 0$. For each defining attribute occurrence there is exactly one rule in R(p) that determines how to compute the value of this attribute occurrence. The evaluation rule defining attribute occurrence $X_{p,k}.a$ has the form: $X_{p,k}.a = f(X_{p,k_1}.a_1, \ldots, X_{p,k_m}.a_m)$.

The example in *Figure 1.4* gives an AG for computing the *type* of a simple expression:

In the example, the "type" of *addexpr* is *real* if the first or the second *expr* has a *real* type; otherwise it is *int*.

If we supplement the derivation tree with the values of attribute occurrences, we get an attributed derivation tree.

Each attribute occurrence is calculated once and only once. The attributed derivation tree for the expression $3 * (2.5 + 4)$ is shown in *Figure 1.5*. The main task in AG is to calculate these attribute occurrences in the attributed derivation tree; and this process is called attribute evaluation. Of course, there are several ways of evaluating the attributed tree. In the case of a simple attribute grammar the occurrences can be evaluated in one Left-to-Right pass; however, there may be more complex grammars where more passes are required. The algorithm for a

```
(1) expr     -> num
      expr.type=num.type;
(2) expr     -> multexpr
      expr.type=multexpr.type;
(3) expr     -> addexpr
      expr.type=addexpr.type;
(4) addexpr  -> expr "ADD" expr
      addexpr.type=(expr[1].type=="real"||
                    expr[2].type=="real")? "real":"int";
(5) addexpr  -> expr "SUB" expr
      addexpr.type=(expr[1].type=="real"||
                    expr[2].type=="real")? "real":"int";
(6) multexpr -> expr "MUL" expr
      multexpr.type=(expr[1].type=="real"||
                     expr[2].type=="real")? "real":"int";
(7) multexpr -> expr "DIV" expr
      multexpr.type="real";
(8) num      -> INTNUM
      num.type = "int"
(9) num      -> REALNUM
      num.type = "real"
```

Figure 1.4: An example of calculating expressions using semantic rules

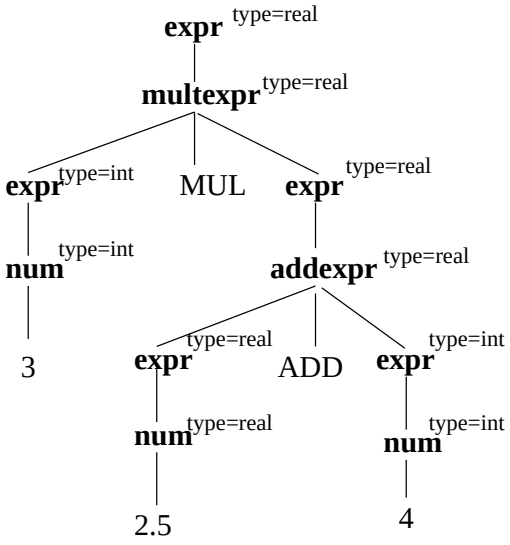


Figure 1.5: Attributed Derivation Tree

simple Left-to-Right pass evaluator can be seen in *Figure 1.6.* This algorithm evaluates the inherited attributes of the root's children recursively. When all inherited attributes have been evaluated, the synthesized attributes will be processed.

Attribute grammars can be classified according to the evaluation method used. The S-Attributed Grammar contains only synthetized attributes. L-Attributed

```
function evaluate_L(node r) begin
   children = left_to_right_list_of_children_of(r)
   while not_empty(children)
   begin
     c = first(children)
     c_inh = inherited_attributes_of(c)
     evaluate_attributes(c_inh)
     evaluate_L(c);
     remove_first_from_list(children)
   end
   r_synth = synthetized_attributes_of(r)
   evaluate_attributes(r_synth)
end
```

Figure 1.6: A simple Left-to-Right evaluator.

Grammars can be evaluated in a single Left-to-Right pass, while a more complex AG may need several Left-to-Right or Right-to-Left passes or even more compress passes to evaluate all of its attributes.

1.1.3 The relationship between XML and Attribute Grammars

Examining the DOM tree and the attributed derivation tree in *Figure 1.3* and *Figure 1.5*, it can be seen that XML documents are noticeably quite similar to attributed derivation trees. There is also a good analogy between AG and XML documents (see Table 1.1). In Attribute Grammars, the Nonterminals correspond to the elements in the XML document.

Attribute Grammars	**XML**
nonterminal	element
formal rules	element specification in DTD
attribute specification	attribute specification in DTD
semantic functions	—

Table 1.1: Analogy between AG and XML

Syntactic Rules are given as an element type declaration in the DTD of the XML file. An attribute specification in the AG corresponds to an attribute list declaration in the DTD. However, there is an important concept in Attribute Grammars which has no XML counterpart, namely semantic rules. It might be useful to apply these semantic rules in the XML environment as well. This would be advantageous as the attribute instances and their values are stored directly in the XML document files. If it were possible to define semantic rules, it would

be sufficient to store the rules that apply to specific attributes; then their correct values could be calculated. With this, it would then be possible to avoid having to store those attributes which could be calculated. In the future, the definition of semantic rules will be an integral part of XML document files.

1.2 XML Semantics Extension

There are several possible approaches for defining semantic rules. One solution would be to use a DTD file. The problem with this is that the DTD cannot be expanded to store the rules in a structured format. The other problem with the DTD file is that the elements are defined using regular expressions, making it rather hard to reference each item separately.

Another approach might be to introduce a new metalanguage that has its own parser. This is an ideal solution as it provides the freedom to add complex semantic rules. These semantic rules have to be stored somehow, since we are using an XML environment. Hence, it surely a sensible idea to store the semantic rules in an XML-based format as well.

1.2.1 Specifying semantics rules: The SRML Metalanguage

We will define a metalanguage called SRML (Semantics Rule Meta Language) to describe semantics rules, which has an XML-based format. The corresponding DTD of this language is listed in Appendix A. The XSD version of it can be found in Appendix B.

A DTD can be viewed as a formal grammar [36]: elements will be nonterminals and the descriptions of the element will be formal rules. Now we would like to define some semantics rules for these formal rules.

The meaning of the elements of SRML are:

semantics-rules : This is the root element of SRML.

rules-for : This element gathers together the semantics rules of a formal rule. In a DTD there is only one working description of an element so the formal rule is determined by that element, which is on the left of the formal rule. This is the *root* attribute of the *rules-for* element.

rule : This element describes a semantics rule. We have to specify which *attrib*ute of which *element* we are going to define in this semantics rule and its value in *expr*. If the value of the *element* is "*srml:root*", then we define an attribute of the root element.

expr : An expression can be a binary-expression (*binary-op*), an *attribute*, a directly defined value (*data* or *no-data*), a conditional expression (*if-expr*,*if-all* or *if-any*), a syntax-condition (*if-element* and *position*) or an external function call (*extern-function*).

if-element : In a DTD element description one can specify a regular expression (with maybe +,*,?, etc. symbols). This element provides us with the possibility of testing the actual form of the input. It contains two *expr* elements. As an expression, the value of the *if-element* is true or false depending on the following: the name of the first *expr*th child (element) in the actual rule equals the value of the second *expr*. The *from* attribute can specify which direction to operate. In other words, it is possible to take an index from the end without actually knowing how many children the parent has.

binary-op : This element is an ordinary binary expression.

position : Returns a 0-based index which identifies the current element's position, taking into consideration the *element* attribute. Possible directions are as follows : begin, end. It is possible to use the *srml:all* identifier, in which case the index returned will be the actual overall position in the DOM tree level. If an *element* name is specified then the returned index will be n, where the element has $n-1$ predecessors or successors with the same element name.

attribute : The attribute is determined by its *element*, *attrib*, *from* and *num* attributes. In the actual rule, this is the *num*th element, where the name matches the value of the *element* (if it is "*srml:any*" it can be anything; if it is "*srml:root*", then it is an attribute of the root) *from* the *begin*ning, the *current* element or the *end*ing. If the attribute does not exist it will be handled as *no-data*.

if-expr : This is an ordinary conditional expression. The first *expr*ession is the condition and it depends on whether if it is true (not zero) or not. The value of the *if-expr* will be the value of the second or third *expr*ession.

if-all : This is an iterated version of the previous *if-expr* expression. The first *expr* is computed for all matching attributes (each selected by *element* and *attrib*ute, which can take a concrete value or "*srml:all*)". We can refer to the value of this attribute using the element *current-attribute*. If the condition (first *expr*) is true for all matching attributes, the value of it is the value of the second *expr*; otherwise it is the third *expr*.

if-any : This is almost the same as the previous one except that it is sufficient that the condition be true for at least one matching attribute.

current-attribute : This is the loop variable of *if-any* and *if-all* elements.

data : This element has no attribute and usually contains a number or a string.

no-data : This element means that this attribute cannot be computed – it is often present in some branches of conditional expressions.

extern-function : This element makes an external function call handled by the implementation. It makes SRML easily extendable.

param : This describes a parameter of an *external-function*.

An SRML definition has to be *consistent*. This means that an attribute instance can only have one corresponding rule. In SRML, an attribute is either inherited (*Figure 1.7(a)*) or synthesized (*Figure 1.7(b)*) like that of Attribute Grammars; hence in the SRML language it is not permitted to have two separate rules for the same attribute.

```
<rules-for root="A">
 <rule element="B" attrib="x">
  <expr>
   <attribute element="srml:root" attrib="x"/>
  </expr>
 </rule>
</rules-for>

Rule: B.x = A.x
```

```
<rules-for root="A">
 <rule element="srml:root" attrib="y">
  <expr><binary-op op="add">
   <attribute element="srml:root" attrib="x"/>
  </expr>
  <expr><data>3</data></expr>
  </binary-op></expr>
  </rule></rules-for>

   Rule: A.y = A.x+3
```

Figure 1.7: (a) An inherited attribute (B.x) (b) A synthesized attribute (A.y)

We will provide an example rule set to demonstrate the advantages of using the SRML language. The rule set in *Figure 1.8* defines the same semantic rules as those stated in *Figure 1.4*. The example only covers the *addexpr* type calculation, as all other elements can be calculated using similar rules.

Now there are a few comments that should be made about *Figure 1.8*. The first is that the rule set is much bigger than that for AG rules. The reason is that this is already in an interpreted form, whereas the AG rules have to be interpreted first. Another thing that has to be explained is the "from" and "num" attributes in the <attribute/> tag. Although the complete list of attributes is given above, we will explain these two attributes, as they are important for this example. The "from" attribute specifies which direction the attribute value is taken from, and "num" is the offset number. So the following term means that we are referring to the *"type"* attribute of the first *expr* element:

```
<attribute element="expr" from="begin" num="1" attrib="type"/>
```

```
AG rule:
(4) addexpr -> expr "ADD" expr
addexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";

SRML rule:
<rules-for root="addexpr"> <rule element="srml:root" attrib="type">
  <expr>
    <if-expr>
    <expr>
      <binary-op op="or">
        <expr>
        <binary-op op=equal>
          <expr><attribute element="expr" num="1" attrib="type" from="begin"/></expr>
          <expr><data>real</data></expr>
        </binary-op>
        </expr>
        <expr>
        <binary-op op="equal">
          <expr><attribute element="expr" num="2" attrib="type" from="begin"/></expr>
        <expr><data>real</data></expr>
        </binary-op>
        </expr>
      </binary-op>
    </expr>
    <expr><data>real</data></expr>
    <expr><data>int</data></expr>
    </if-expr>
  </expr>
</rule> </rules-for>
```

Figure 1.8: An SRML example for "type" attribute of the addexpr element

1.3 SRML description types

As we mentioned in Section 1.1.2, the attribute grammars can be classified according to the evaluation method employed [2].

By analogy we could introduce S-, L-, ASE-,... SRML descriptions. Here, we will only define S- and L-SRML descriptions.

Actually, there are only two relevant factors that we need to know in the SRML description to decide whether it is an S/L-SRML description. The first one is the set of defined attributes associated with the rules, while the second is the set of referenced (usable) attributes in these definitions.

1.3.1 The S-SRML description

S-attributed grammars are the simplest attribute grammars: they have only synthesized attributes. As in XML, in SRML we do not distinguish between synthesized attributes and inherited ones. In this environment we can define an S-SRML description, in analogy with S-attributed grammars:

definable attributes : For each rule we can only define the attributes of the (srml:)root nonterminal (the element that is on the left) because synthesized attributes are only definable in a rule if the root element contains them.

usable attributes in definitions : All attributes in this rule presume that there are no circular dependencies.

1.3.2 The L-SRML description

An L-attributed grammar can contain synthesized and inherited attributes, but the dependencies between them must be evaluated in one left-to-right pass. In the SRML environment it means the following:

definable attributes : All available attribute occurrences, keeping consistency (see Section 1.2.1) in mind.

usable attributes in definitions : We use one left-to-right pass to evaluate the attributes. In a rule environment we first calculate the attributes of the children nonterminals, and after the attributes of the root nonterminal in a suitable order. An SRML description is called an L-SRML description if there is a suitable order in attributes which carries out the following: if there is an attribute reference in the definition of an attribute, then the value of referenced attribute has already been calculated.
To be more precise:

- In the definition of an attribute of the root there can be attributes of any children, or those attributes of the root that have been defined earlier in the SRML description.
- In the definition of an attribute of a child there can be attributes of any children which lie to the left of it, or are those attributes of the same child that have been defined earlier in the SRML description.

1.3.3 Compacting / Decompacting

Using the evaluation methods outlined earlier it is now possible to describe how they can be used in the compaction of XML documents. Before going into detail on how the method is built up some definitions are needed for *Figure 1.9*.

The "Complete XML Document" refers to the document in its original size and form. The "Reduced XML Document" is the output of the compaction, which of course has a smaller size relative to the original. The "Semantic Rules" are the SRML rules used to compact and decompact the XML document. In *Figure 1.9*,

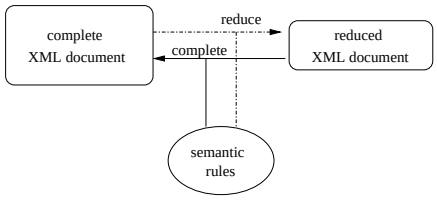


Figure 1.9: The Compacting/Decompacting method

it is clear that the *decompacting (complete)* procedure expands the document and recreates the original XML document.

The *compacting (reduce)* procedure does just the opposite. The input used is a complete XML document with an attached semantic rules file. Every attribute which can be correctly calculated using the attached rules will be removed. This results in a reduced, compacted XML document. It is important to note that if the semantic rule for a given attribute does not give the correct value the attribute is not removed, thus maintaining the document's integrity.

Now we apply the rule set described in *Figure 1.8* to the input XML shown in *Figure 1.4*, also including the additional rules, which are very similar. The compacted XML can be seen in *Figure 1.10*.

```
<expr>
  <multexpr op="mul">
    <expr><num type="int">3</num></expr>
    <expr>
      <addexpr op="add">
        <expr><num type="real">2.5</num></expr>
        <expr><num type="int">4</num></expr>
      </addexpr>
    </expr>
  </multexpr>
</expr>
```

Figure 1.10: The compacted XML of the expression

The size of this XML is considerably smaller than the size of the input (*Figure 1.1*). The only "type" attribute that was kept was the *num* element's "type" attribute. The reason was that, if we have the "type" of *num*, then we can calculate the *expr* "type". Once we have the *expr* "type" the *addexpr* and *multexpr* "type" attributes can be calculated as well. This is why there is no sense in storing all the types if only the *num* "type" is needed.

After the XML document has been compacted, it can then be compressed using a standard compressor like gzip or an XML compressor like XMill. An example of a compacted XML is shown in *Figure 1.10*.

The file containing the SRML description is an extension of the DTD. If the XML files were to be compressed instead of compacted this possibility would be lost. One can say that this method of compacting makes the XML format lose its self-describing ability, but the size reduction we gain by removing attributes and the fact that it still is readable by any XML viewer and can be further compressed by any XML compressor, makes this sacrifice worthwhile.

1.4 SRMLTool: a compactor/decompactor for XML documents

We implemented a tool for the compaction of XML documents. This tool uses SRML to store semantic rules, describing the attributes of the XML documents. It means that the inputs of the tool are an XML document and an SRML file to define semantic (computation) rules for some attributes of the document. The output may be a reduced document (some attributes are removed) or a complete document (the missing attributes with corresponding rules have been calculated and restored). *Figure 1.11* shows the modular structure of the tool.

The implementation uses a general evaluator. Currently, attribute occurrences are evaluated depth first using Left-to-Right passes. This is a multipass algorithm where the algorithm described in *Figure 1.6* is executed several times. During each pass a separate attribute occurrence is evaluated. The implementation of the algorithm starts off by reading both the XML file and SRML file into separate DOM trees. This saves a lot of time on operations performed later. Then the XML DOM tree is processed using an inorder tree visit routine that examines every node and every attribute. The purpose of this examination is to find out which attributes have corresponding SRML rules. Unfortunately this approach has high resource requirements. A way of optimizing it is to allow only L-SRML rules (see *Section 1.3.2*) and use a SAX parser.

1.4.1 The Reduce algorithm (compacting)

This algorithm removes those attributes that can be correctly calculated using the SRML semantic rules. Every attribute can have only one rule in the SRML rule set. If there are more rules for the same attribute, the first one will be used. There is a possibility that the attribute cannot be correctly calculated, which means that the value of the attribute differs from the value given in the rule. In this case the attribute will not be removed, since the value is different. During the implementation of the *Reduce* (compacting) algorithm, the function of the *Complete*

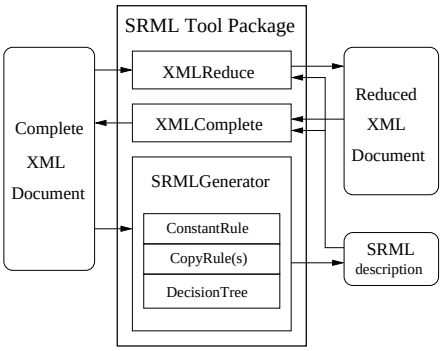


Figure 1.11: The structure of the implementation

(decompacting) algorithm had to be considered. If there is a rule for a non-defined[1] attribute it has to be marked somehow. If this is ignored, the *Complete* algorithm will insert a value for it and make the *compaction/decompaction* process inconsistent. To remedy this problem, the *srml:var* attribute is introduced into the *compacted* document. This attribute marks the name of those attributes that were not present in the original document. Consider, for example, the following SRML rule set:

```
<rules-for root="A">
 <rule element="srml:root" attribute="x">
  <expr><data>100</data></expr>
 </rule>
 <rule element="srml:root" attribute="y">
  <expr><data>200</data></expr>
 </rule>
</rules-for>
```

Suppose the input XML document is the following:

```
<calc>
 <A x="100" y="200"/>
 <A x="99" y="200"/>
 <A x="88" y="201"/>
 <A/>
</calc>
```

When applying the rules to the input XML document, the result of compaction (*Reduce*) will be the following:

```
<calc>
 <A/>
 <A x="99"/>
 <A x="88" y="201"/>
 <A srml:var=" x y "/>
</calc>
```

[1]Non-defined attributes can occur if there are IMPLIED attributes in the XML file.

The example shows that in the first A element both x and y attributes were removed as their value matches the SRML rule value. In the second, only the y attribute could be removed since the x attribute had a different value. In the third A element, both x and y attributes were kept as their values differed from the SRML rule values. In the last A element a new attribute called *srml:var* had to be inserted because the input XML document had neither attribute in this element. If the attributes had not been marked, then the decompacting would have added both attributes with the value mentioned in the SRML rules.

Our implementation can reduce XML documents even when there is a circular reference in the rules. For example if the rules $A.x = B.x$, $B.x = C.x$ and $C.x = A.x$ (*Figure 1.13*) are given there is an exact rule for each attribute, but only two of them can be deleted if it needs to be restored later. The following algorithm is used to remove the attributes (this algorithm can also resolve circular dependencies):

1. Create a dependency list (every attribute may have input and output dependencies, the input dependencies being those attributes on which it depends and the output are those that depend on it. The dependencies are represented as edges)

2. If the list is empty, then goto 5. Look for an attribute that has no input or output edges. If there is one, then it can be deleted and removed from the list (since this can be restored using the rule file), next goto 2. If there isn't any list, goto 3.

3. Look for an attribute that has only output edges. This means that other attributes depend on this attribute (e.g: the A.x attribute in *Figure 1.12*). Delete this attribute and the output edges, remove it from the list, then goto 2. If there isn't one, goto 4.

4. Check for circular references. This is the last possible case since if all the attributes remaining in the dependency list have both input and output edges it means that the attributes are transitively dependent on each other (*Figure 1.13*). Select the first element in the list (this will serve as the basis for the circular reference). Keep this attribute and remove it from the list (keeping it means that it is added to a *keep* vector). Goto 2.

5. END

The algorithm always terminates since the dependency list is always emptied. The algorithm has a vector which contains those attributes that will not be removed. Note, however, that this algorithm is not the most optimal solution, but it is reduction safe and the completion process can be performed without any loss.

Figure 1.12: Simple dependency

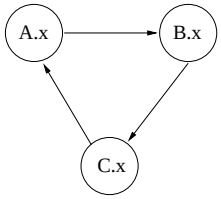


Figure 1.13: Circular dependency

1.4.2 The Complete algorithm (decompacting)

This algorithm restores the attributes which have corresponding SRML semantic rules. The attributes which were marked with the *srml:var* attribute will not be restored (see *Section 1.4.1*). In contrast to the traditional evaluator where we define which attribute is evaluated in which pass our approach tries to evaluate every attribute occurrence during each pass using an attribute occurrence queue. As mentioned in *Section 1.4*, a DOM tree is built from the XML file. After this an inorder tree visitor is called to find out which attributes have corresponding SRML rules. If an attribute having an SRML rule is found it is stored in a vector that is later processed. This vector is used for decompacting, which is a two-stage operation. First a vector is created with those attributes having corresponding rules, then in stage two the vector elements are processed. This speeds up the decompacting since the DOM tree is visited only once. Afterwards tree pointers are used to access the nodes.

1.5 Learning SRML rules

In some cases the user does not know the relationship among the attributes of an XML document so therefore he cannot provide SRML rules. The input of the module is the XML document which needs to be analyzed. The output will be a set of rules optimized for the input which enable the compaction. The *SRMLGenerator* module is based on a framework system so that it can be expanded later with plug-in algorithms. Every plug-in algorithm must fit a defined interface.

The process of learning is as follows:

1. Read the input XML file.

2. Enumerate the plug-in algorithms and execute them sequentially.

3. After a plug-in has finished, it marks those attributes for which it could find an SRML rule to in the DOM tree, making the next algorithm only process those attributes that have no rules.

4. If all the plug-ins have been executed, then write the output SRML rule file; otherwise continue processing the XML file.

Figure 1.14 shows the process of learning. The generated rules are concatenated one after each other, creating a single SRML rule file at the end of the process.

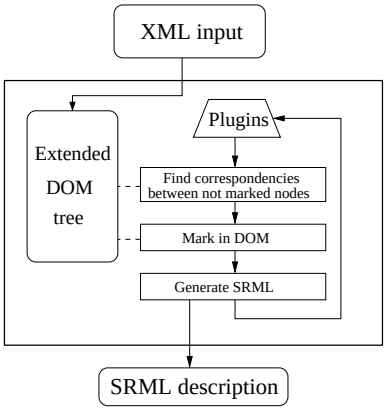


Figure 1.14: Learning SRML Rules

SRML files have other crucial uses apart from making the XML files more compact. One of these is that SRML allows the user to discover relationships and dependencies among attributes, which may not have been seen by the user. In this case of course, the SRML file has to be created dynamically via machine learning and other statistical methods. The SRML files created by machine learning can be used as an input to other systems such as decision-making systems, where the relationship between specific criteria is examined. It may be employed in Data Mining and other areas where relationships in large amounts of data are sought.

1.5.1 The SRMLGenerator module's learning algorithms

The *SRMLGenerator* currently contains five plug-in algorithms. These algorithms can be expanded with additional plug-in algorithms thanks to our framework system. A new plug-in algorithm can be created simply by creating a class which conforms to the appropriate interface. The execution order of the plug-ins is crucial, since the order defines the efficiency of the compaction and also the execution

time. During the testing phase we found the following order to be the most effective (an observation based on experiment):

1. SRMLCopyChildRule
2. SRMLCopyAttribRule
3. SRMLCopyParentRule
4. SRMLDecisionTree
5. SRMLConstantRule

Below, we will describe each learning module in detail.

SRMLConstantRule

This is the simplest learning algorithm in the package. This algorithm uses statistical analysis to retrieve the number of attribute occurrence values and then decides whether to make a rule for it. For instance this algorithm searches for $B.x = 4$ and $A.B.x = 4$ type of rules. The difference between these two types is that the first is synthesized while the second is inherited (see *Figure 1.15(a)*).

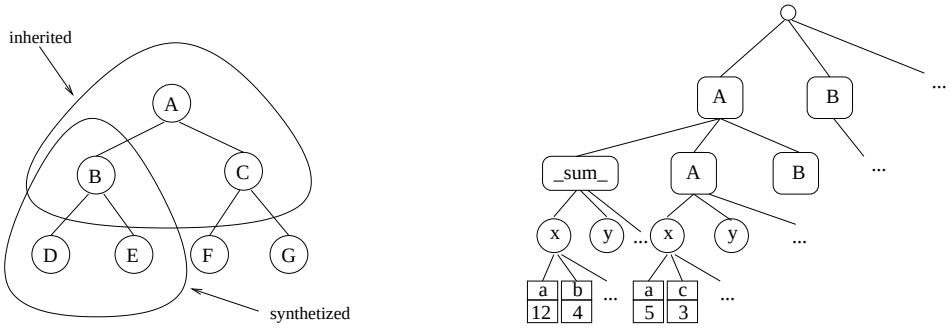


Figure 1.15: (a)The two contexts of a Node (b)The statistical tree of the SRMLConstantRule

The decision is based on whether the size of the new rule would be bigger than that of the size decrease achieved by removing the attributes. The tree in *Figure 1.15(b)* is used in evaluations performed by the algorithm.

To get a clearer understanding of the tree, a brief explanation will be provided of how it is built. First the input XML file is parsed and each attribute occurrence is examined. All occurrences have two counters incremented in the tree: the

attribName.value of *elementName* (synthesized case) and one of *parentElementName* (inherited case).

After this stage the exact benefit of generating SRML rules in a synthesized or inherited form can be calculated using the statistical tree created. The better one will be chosen (if a rule can be generated).

Copy Rules

These algorithms search for $A.x = B.x$ rules. The time and memory requirements of searching for this type of rule in one stage can be rather high. That is why the implementation was separated into three modules: *SRMLCopyChildRule* ($x = B.x$) *SRMLCopyAttribRule* ($x = y$) and *SRMLCopyParentRule* ($B.y = x$). The implementation uses similar statistical trees as those mentioned above. Here, the algorithm determines whether the attribute was inherited or synthesized. Inherited attributes are handled by the *SRMLCopyParentRule* module,whereas synthesized attributes are handled with the *SRMLCopyChildRule* and *SRMLCopyAttribRule* modules. In *Figure 1.16*, examples can be seen for each type. In the case of (1), the *book.firstSection* is equal in value to the *section.name*, making this attribute a synthesized one. In (2), the *word* attribute of the book matches the total attribute, thus making it a synthesized attribute as well. In (3), *section.author* is the same as the *book.author* so it is an inherited attribute. In the example, there are parts where the attributes do not match. This is permitted since it is not obligatory that every attribute with the same name match.

SRMLDecisionTree

The *SRMLDecisionTree* plug-in is by far the most advanced of the currently implemented algorithms. It makes use of a machine learning approach [34] in order to discover relationships and builds if-else decisions using a binary tree similar to the ID3 [37] algorithm. Next, we elaborate on how the *SRMLDecisionTree* plug-in actually works.

The algorithm first narrows the set of attributes for which rules need to be made. The criteria of this "narrowing" is that the given attribute must have at least two different values whose occurrences are *dominant*. The meaning of *dominant* in this case is that the attribute value occurs frequently enough in the input, making it an excellent target for compaction. The operation of finding *dominant* attributes is performed though a function which can be altered. Currently the algorithm considers an attribute value *dominant* if the value occurrence exceeds 5. Although we defined the dominance limit to 5, it does not necessarily mean that this is a hard limit. The dominance function can be replaced at will. If the number of *dominant* attribute values is less than two, it is not a problem since the

```
1. SRMLCopyChildRule XML: <book firstSection="Introduction">
  <section name="Introduction" word="200"/>
  <section name="Summary" word="400"/>
</book>

SRML:
<rules-for root="book">
 <rule element="srml:root" attrib="firstSection">
  <expr><attribute element="section" attrib="name"/></expr>
 </rule>
</rules-for>

2. SRMLCopyAttribRule XML:
<book word="200" total="200">
 <section word="200"/>
</book>

SRML:
<rules-for root="book">
 <rule element="srml:root" attrib="word">
  <expr><attribute element="srml:root" attrib="total"/></expr>
 </rule>
</rules-for>

3. SRMLCopyParentRule XML:
 <book author="James West">
 <section author="James West" title="Introduction">
 <section author="Mary Jones" title="Summary">
</book>

SRML:
<rules-for root="book">
 <rule element="section" attrib="author">
  <expr><attribute element="srml:root" attrib="author"/></expr>
 </rule>
</rules-for>
```

Figure 1.16: Examples of the Copy rules

next plug-in algorithm (in this case the *SRMLConstantRule*) will try to find a rule for the attribute. The reason why the attribute value number is limited to two is that with two different dominant values if-else branches can be created, which in turn allows a more optimal rule generation process.

We generate a learning table for each "narrowed" *dominant* attribute. To demonstrate how these tables are created and what their contents are, a simple example will be provided. The example creates a car-pool database for cars. Each car can have the following attributes: colour, ccode (colour code), doors, type. The XML format of the example is shown in *Figure 1.17*.

Based on the input XML file described in *Figure 1.17*, two learning tables will be generated, assuming that there are enough dominant attributes. At this stage of the processing it is not known whether the attribute is synthesized or inherited,

```
<car-set num="100">
 <car color="blue" ccode="5" doors="5" type="minivan"/>
 <car color="red" ccode="3" doors="3" type="compact"/>
 <car color="blue" ccode="5" doors="5" type="standard"/>
 <car color="green" ccode="2" doors="5" type="standard"/>
 <car color="black" ccode="1" doors="3" type="convertible"/>
 <car color="blue" ccode="5" doors="3" type="compact"/>
 <car color="green" ccode="2" doors="3" type="convertible"/>
 ...
</car-set>
```

Figure 1.17: An example of a decision tree input

so learning tables are generated for both contexts. For the synthesized case (*Table 1.2*) only one table is created, but for the inherited case (*Table 1.3*) the number of tables depends on how many parents the attribute's element had in the whole input file.

.find[color]	ccode	doors	type
blue	5	5	minivan
red	3	3	compact
green	2	5	standard
black	1	3	convertible
blue	5	5	standard
blue	5	3	compact
green	2	3	convertible
...			...

Table 1.2: A synthesized learning table for *Figure 1.17*

.find[car.colour]	car.ccode.1	car.doors.1	car.type.1	...
blue	5	5	minivan	...
red	3	3	compact	...
green	2	5	standard	...
black	1	3	convertible	...
blue	5	5	standard	...
blue	5	3	compact	...
green	2	3	convertible	...
...				...

Table 1.3: An inherited learning table for *Figure 1.17*

The headers of the columns are the names of the attributes which are present in the rule contexts (inherited, synthesized) of the current attribute. If an attribute occurs more than once in the rule contexts, a new column is appended for each new attribute occurrence.

The number of rows depend on how many rule contexts there are that contain this attribute. The values of the fields are the values that the attribute takes in a

given context. If the attribute is not present in a context, the value is marked by a minus (-) sign.

The learning tables created by this algorithm can be used as inputs for external learning systems. The learning algorithm builds up a binary tree similar to that for the ID3 algorithm with a depth limit of 3. The reason why we use a binary tree is that the current specification of the SRML format only handles if-else statements and it would be rather difficult to code a multiple branch tree.

With every table, a new rule is learned for the given attribute. Next, the algorithm decides (based on the generated tree) whether to select the synthesized rule or some of the inherited rules or neither. This decision is made using the information stored in the leaves of the tree. This information helps the algorithm effectively decide how many attributes will be eliminated during compaction if the given rule is added to the rule set. The algorithm will select those rules that achieve the maximum compaction.

At the end of the algorithm, the SRML rules are generated from the selected trees and the appropriate attributes are marked in the input DOM tree. In the case of *Figure 1.17*, the learned rule set will be a relationship between the *ccode* (numeric) and the *colour* (text) attribute.

NOTE: Every plug-in tries to make the optimal decision, the only factor that being currently not considered is the length of the *srml:var* attribute. This is why in some cases the SRMLConstantRule increases the size of the compacted file. The framework detects when a size increase occurs and does not execute the specific learning module.

1.6 Experimental results

In the field of Software Engineering the XML representation is treated as a standard, so it is widely used (e.g. XMI based models). The testing of our implementation was done via CPPML [12], an XML exchange format that is used as an output of the Columbus Reverse Engineering package [39]. However, this method can be applied to any XML domain without any major restrictions.

1.6.1 A real-world case study: CPPML

CPPML files can be created from a CPP file. Here, CPPML is a metalanguage capable of describing the structure of programs written in C++. Creating CPPML files can be performed via the Columbus Reverse engineering package [39] (CPPML is XML-based).

To illustrate how CPPML stores a C++ program, let us consider the following C++ program:

```
class _guard : public std::map<std::string, _guard_info>
{
  public:void registerConstruction(const type_info & ti)
  {
    (*this)[ti.name()]++ ;
  }
  ...
};
```

The CPPML form of the program could be the following:

```
<class id="id20097" name="_guard" path="D:\SymbolTable\CANGuard.h" line="71"
 end-line="90" visibility="global" abstract="no"defined="yes" template="no"
 template-instance="no" class-type="class">
   <function id="id20102" name="registerConstruction" path="D:\SymbolTable\CANGuard.h"
    line="75" end-line="76" visibility="public" const="no" virtual="no" pure-virtual="no"
    kind="normal" body-line="75" body-end-line="76" body-path="D:\SymbolTable\CANGuard.h">
        <return-type>void</return-type>
        <parameter id="id20106" name="ti" path="D:\SymbolTable\CANGuard.h" line="74"
         end-line="74" const="yes">
        <type>type_info&amp;amp;</type>
        </parameter>
    </function>
    ...
</class> ...
```

In the CPPML definition, a lot of attributes can be calculated or estimated using other attributes. One of these is the *kind* attribute, which stores the type of the function. This *kind* can be *normal*, a *constructor* or *destructor*. If the function name matches that of the class name, then it is a *constructor*; if the function name starts with a $\sim$, then it is a *destructor*.

Expressed in SRML form, this might look like the following:

```
<rules-for root="class">
  <rule element="function" attrib="kind">
    <expr>
      <if-expr>
        <expr>
          <binary-op op="equal">
            <expr><attribute attrib="name"/></expr>
            <expr><attribute attrib="name" element="srml:root"/></expr>
          </binary-op>
        </expr>
        <expr><data>constructor</data></expr>
        <expr>...</expr>
      </if-expr>
    </expr>
  </rule>
</rules-for>
```

It is not necessary to provide precise rules, since the *Compact* algorithm will only remove those attributes that can be correctly calculated from the rules. Consider the following estimation:

1. A function declaration starts and ends on the same line.

2. The implementation of a class's function is usually in the same file as the previous function's implementation.

3. The parameters of a function are usually in the same file, perhaps somewhere in the same line.

4. The visibility of the class members is normally the same as that for the previously defined members.

Expressed in SRML form, these "estimated" SRML rules may look like the following:

```
<rules-for root="function">
  <rule element="parameter" attrib="end-line">
      <expr><attribute attrib="line"/></expr>
  </rule>
  <rule element="parameter" attrib="line">
    <expr><attribute attrib="line" num="-1"/></expr>
  </rule>
  <rule element="parameter" attrib="path">
    <expr><attribute attrib="path" num="-1"/></expr>
  </rule>
</rules-for>
```

After running the *compaction* module, the following XML document is produced:

```
<class id="id20097" name="_guard" path="D:\SymbolTable\CANGuard.h" line="71"
 end-line="90" visibility="global" abstract="no" defined="yes" template="no"
    template-instance="no" class-type="class">
  <function id="id20102" name="registerConstruction" line="75" end-line="76" visibility="public"
   const="no" virtual="no" pure-virtual="no" kind="normal" body-path="D:\SymbolTable\CANGuard.h">
    <return-type>void</return-type>
    <parameter id="id20106" name="ti" line="74" const="yes">
      <type>type_info&amp;amp;</type>
    </parameter>
  </function>
  ...
</class>
```

Using the rules described above, it produced a compaction ratio of 68.9%, since the original fragment was 2,180 bytes and the compacted was 1,502 bytes. This ratio can be further improved with the introduction of new SRML rules.

1.6.2 Compacting CPPML with SRML rules created by hand

We implemented the rules in SRML (mentioned in the previous section) and applied them to three different examples. These were:

symboltable (399KB): one of the source files of the Columbus system

jikes (2233KB): the IBM Java compiler

appwiz (3546KB): a base application generated by Microsoft Visual Studio's AppWizard

The results achieved using SRML files created by hand are shown in *Table 1.4*. The input files here were in CPPML form. The (C) bracket indicates that the compressors were applied to the compacted version of the XML file.

File	SymbolTable	Jikes	AppWiz
Original	399 321	2 233 824	3 547 297
gzip [ratio]	30 460 [7.62 %]	177 051 [7.92 %]	244 174 [6.68 %]
XMill [ratio]	19 786 [4.95 %]	114 275 [5.11 %]	145 738 [4.10 %]
Compacted [ratio]	296 193 [74.10 %]	1 736 267 [77.70 %]	2 238 308 [63.10 %]
gzip(C) [ratio]	26 308 [6.58 %]	160 609 [7.18 %]	206 522 [5.82 %]
XMill(C) [ratio]	18 008 [4.50 %]	108 458 [4.85 %]	134 217 [3.78 %]

Table 1.4: Compaction table using handwritten rules

Although the compaction ratio is smaller than that achieved by compression, when compaction and compression are combined, the method can improve the efficiency of the compressor. Applying XMill on the compacted SymbolTable led to an overall size reduction of about 10%, since XMill compressed the original SymbolTable to 19,786 bytes, whereas applying XMill after the file had been compacted resulted in a size of 18,008 bytes. This 10% efficiency increase can be very useful for embedded systems.

Manual rule generation isn't a hard task for a domain expert who knows the system he wishes to apply the method to, since he knows the attributes and this can create the appropriate rule sets. Creating hand written rules for the CPPML environment took less than an hour.

1.6.3 Compacting CPPML with machine learning SRML rules

In *Table 1.5*, a comparison is made between the efficiency of the machine learned and hand-generated SRML rules. The percentage scores shown in the *Diff* field show how big the difference was compared to the original file size.

In some cases the compaction ratio achieved using SRML files generated via machine learning can attain those of hand-generated SRML files (e.g. *Jikes*). However, creating hand-generated rules requires time and effort, since the user needs to know the structure of the XML file. The machine learning approach takes this burden off the user and generates rules automatically. These results can be improved by adding new plug-in algorithms into the *SRMLGenerator* module like

Filename	Manual	Machine	Diff
SymbolTable (399 321)	296 193 [74.10 %]	313 873 [78.60 %]	17 680 [4.42 %]
Jikes (2 233 824)	1 736 267 [77.70 %]	1 737 872 [77.79 %]	1 605 [0.07 %]
AppWiz (3 547 297)	2 238 308 [63.10 %]	2 589 526 [73.00 %]	351 218[9.90 %]

Table 1.5: Comparison of machine learned and hand written rules

those using advanced decision making systems, other types of statistical methods and concatenation parsers (which search for relationships among concatenated attribute values).

Using Machine Learning to generate rules can be costly, but it is sufficent to generate the rules once; then they can be reused over time for each XML file in the given domain.

Since the execution order of the plug-ins really matters (an effective order was described in *Section 1.5.1*), this is why using the Copy rules (*SRMLCopyChildRule, SRMLCopyAttribRule, SRMLCopyParentRule*) seems initially to provide the optimal solution, which is an observation based on experiment. First, the copy rules are processed. The order of these is actually not important, but they offer simple relationships which cover more attribute occurrences. The reason why using the SRMLDecisionTree plug-in is applied before last is that it takes a long time to process large files and it is specialized for more complex and specific occurrence discovery; and most attributes can be removed beforehand using other plug-ins. Using *SRMLConstantRule* at the end is useful, since it may remove constant occurrences that were left untouched by the previous plug-ins. In some cases it may be better to choose another order, but this needs to be examined further.

1.6.4 Analyzing the Learning modules

Here, we list the performance of each learning module in the sequence defined in *Section 1.5.1.* A comparison is made by building up the rule set, adding one rule type at a time, then noting the compaction ratio it achieves in practice. This provides a good basis for future amendments. The results are shown in *Table 1.6* below. The sizes include the size of the generated SRML file as well. The order used is listed in *Section 1.5.1.*

As is clear from the table, during the comparison *SRMLConstantRule* decreased the efficiency in *Jikes* and *AppWiz* compaction. The reason for this is that the current cost function does not take into account the case where an attribute is absent in an element. In this case the compactor has to mark this attribute with the *srml:var* attribute (see *section 1.4.1*). If there were many "missing" attributes it is possible that the file size might increase. The framework detects this and does not apply the given learning module. This in many cases is quite

Module	SymbolTable	Jikes	AppWiz
original	399 321	2 233 824	3 547 297
1	359 964 [90%]	2 022 497 [90%]	2 959 427 [83%]
1,2	351 999 [88%]	1 938 549 [86%]	2 889 183 [81%]
1,2,3	346 830 [86%]	1 871 154 [83%]	2 753 721 [77%]
1,2,3,4	337 825 [84%]	1 737 872 [77%]	2 589 526 [73%]
1,2,3,4,5	313 873 [79%]	1 821 228 [82%]	2 773 946 [78%]

Table 1.6: Comparing learning modules

effective. The *SRMLGenerator* module saves the SRML files -after each plugin has been executed- to separate temporary SRML files. This is good for checking the efficiency of each plug-in. Here, this option can be disabled using a command line parameter.

1.6.5 Combining machine learning and manual rule generation

It is possible to combine machine learning and manual rule generation into a single rule set. This is useful when the user knows some relationships among attributes, but not all of them. The module accepts preliminary SRML files as well, meaning that there are some rules in the file but not all. This file is processed and new rules are appended, making it more efficient. Below the efficiency of this method is shown in a table format.

File	Manual	Machine	Combined
SymbolTable (399 321)	296 193 [74.17 %]	313 873 [78.60 %]	281 088 [70.40 %]
Jikes (2 233 822)	1 736 285 [77.72 %]	1 737 872 [77.79 %]	1 367 244 [61.20 %]
AppWiz (3 547 297)	2 238 308 [63.09 %]	2 589 526 [73.00 %]	2 038 569 [57.46 %]

Table 1.7: Combining machine learning and manual rule generation

Table 1.7 shows that combining machine learning with manual rule generation is quite effective. When running XMill on the compacted XML created with the combined rules, an increased compression ratio can be achieved (see *Table 1.8*). When machine learning and manual rule generation are combined, the additional compressibility of the compacted document can be much as 26%. For example in *Table 1.8* AppWiz was compacted to 145 738 bytes, which is 4.1% of the original document. If we first compact the XML using combined rules, then execute the *XMill* on it, the resulting file size is 106 773 bytes, which is 3.01% of the original and the overall file size decrease is 26.73% (this is the size difference between the manually generated then compressed and the combined generation and compressed

file). Since the whole system is based on a plug-in framework it can be readily extended with more efficient machine learning plug-ins.

File	Original	Manual	Combined
SymbolTable	19 786	18 008	17 876
399 321	4.95 %	4.50 %	4.47 %
		8.98 %	9.70 %
Jikes	114 275	108 458	92 102
2 233 822	5.11 %	4.85 %	4.12 %
		5.09 %	19.40 %
AppWiz	145 738	134 217	106 773
3 547 297	4.10 %	3.78 %	3.01 %
		7.90 %	26.73 %

Table 1.8: XMill compression for combined and manual compaction

1.6.6 Resource requirements of the tool

For testing, a Linux environment was used on a PC (AMD Athlon XP 2500+ 512MB DDR). The package requires about 200MB of memory since the DOM tree takes up a lot of space (this point was mentioned in *Section 1.4*). The *AppWiz* file was *compacted* in approximately 2 minutes and decompacted in 30 seconds. The execution time was long in the case of machine learning (*SRMLDecisionTree*) since it had to generate lots of learning tables, which can be rather large at times. Depending on the complexity and level of recursion, the execution time ranged from 2 hours to 30 hours.

Although it is true that learning compacting rules is not very fast, once the rules are generated they can be used over and over again. It is also possible to create a more effective compacting implementation to increase the speed still further.

Trying to determine the running time of the method is not easy, and it can be only estimated. The decision tree learning algorithm used is a standard ID3 algorithm. This algorithm is applied to every attribute that has at least two dominant values. The size of the learning table depends on all of the attribute occurrences and their environments. This is why it would be hard to categorize the running time into standard cubic/quadratic classes; however, it strongly depends on the input file size and the complexity of the relationships contained within.

1.7 Related Work

The first idea about adding semantics to XML documents was mentioned in [36]. The authors furnished a method for transforming the element description of DTD into an EBNF Syntactic rule description. It introduced its own SRD (Semantics

Rule Definition) comprised of two parts: the first one describes the semantic attributes[2], while the second one gives a description of how to compute them. SRD is also XML-based. The main difference between the approach outlined in their article and ours is that we provide semantic rules not just for newly defined attributes but also for real XML attributes. Our approach makes the SRML description an organic part of XML documents. This kind of semantic definition could offer a useful extension for XML techniques. We can also generate the SRML files using machine learning. Our SRML description also differs from the SRD description found in that article. In SRD, the attribute definition of elements with a + or * sign is defined in a different way from the ordinary attribute definition and can only reference the attributes of the previous and subsequent elements. The references in our SRML description are more generic, and all expressions are XML-based.

We are not aware on any study on generating rules for XML files. We came across an article that generates rules for Attribute Grammars, which was introduced in [17]. The idea is to provide a way of learning attribute grammars. The learning problem of semantic rules is transformed into a propositional form. The hypothesis induced by a propositional learner is than transformed back into semantic rules. AGLEARN was motivated by ILP learning and it can be summarized in the following steps:

i) The learning problem of the semantic rules is transformed into propositional form;

ii) A propositional learning method is applied to solve the problem in propositional form;

iii) The induced propositional hypothesis is transformed back into semantic rules;

This method is similar to ours as it learns and uses semantic rules based on examples as training data, but it is only effective on attributes with very small domains. In contrast to our method, it searches for precise rules that can use approximated rules as well.

The reason why these papers are cited here is that they in some way try to accomplish what our algorithm actually accomplishes. They mostly use semantic relations, and some authors even define a separate language (XML based) to describe the operations. It should be emphasized that our algorithm is not a compressor, but a compactor. We do not wish to compare our algorithm with a compressor algorithm, but if we apply a compressor to the compacted document, then we can achieve better results than other standalone compressors. Furthermore, if we generate the SRML file for a group of specific XML documents then it is not necessary to generate SRML rules for each input XML document in that group. This makes its applicability more feasible.

[2]These are newly defined attributes which differ from those defined in XML files.

Below, some compression algorithms are presented, since they can be combined with our approach, and make their overall compression more effective.

In [31], a Minimum Length Encoding algorithm is used. The algorithm operates in a breadth-first order, considering the children of each element from the root in turn. The encoding for the sequence of children of the element with name n in the document is based on how that sequence is parsed using the regular expression in the content model for n in the DTD. This algorithm employs a DTD-centered approach.

Another article that ought to be mentioned is [9], where the authors create a multi-channel access to XML documents. An XML query processor is used to accomplish the structural compression. Lossless and lossy semantic compressors are defined along with an XML-based language for describing possible associations between similar parts of documents and semantic compressors. The main idea is to process the XML document in such a way that elements can be regarded as tuples of a relation. These tuples are structured as a datacube, with aggregate data on suitable dimension intervals. This is a relational approach that treats the XML attributes as relations (formal expressions). The drawback of it is that it has a lossy compression.

XMill [32] is an open source research prototype developed by the University of Pennsylvania and AT&T Labs, building on the gzip library to provide an XML-specific compressor. It uses a path encoder to support the selective application of type-specific (i.e. based on design knowledge of the source document) encoders to the data content. The data and content of the source document are compressed separately using redundancy compression. The idea is to transform the XML into three components: (1) elements and attributes (2) text, and (3) document structure, and then to pipe each of these components through existing text compressors. This is a compressor as well, mostly used by other algorithms. It also makes use of LZW as a compression algorithm.

The idea behind that given in [10] is to use multiplexed hierarchical PPM (prediction by partial match) models to compress XML files. The procedures detailed in the article may in some cases be better than XMill's procedure. The reason is that XMill's base transformation has drawbacks like precluding incremental compressed document processing and requires user intervention to make it efficient. It mentions that MHM (multiplexed hierarchical modelling) can achieve a compression ratio up to 35% better than the other approaches, but it is rather slow.

Another interesting approach for XML compression is described in [41]. This article describes how this tool can be used to execute queries against the compressed XML document. It uses several compression methods like Meta-data compression, Enumerated-type Attribute Value compression, Homomorphic compression. Using these techniques, it creates a semi-structured Compressed XML

document, which is binary, but retains query capabilities. It creates a frequency table and a symbol table, that are than passed on to the XGrind kernel. This approach is better then using completely binary chunks of data, as it can query the file.

1.8 Summary

We introduced a semantics extension of XML and designed a metalanguage called SRML (Semantic Rule Meta Language) that can store the semantics definition of XML attributes.

We presented the most important application of the XML semantics extension: that of compaction. We also combined this compaction with general and XML specific compressors, and achieved a relative compression improvement of 20-30%.

We used simple machine learning method to create semantic rules automatically, and also described how to combine an expert-generated SRML with an automatic machine learning procedure.

The basic idea behind the method and introducing the metalanguage and its main applications are my own results, which were published in [20]. The compression application and the learning framework published in [26] are joint results with Miklós Kálmán. The XSD adaptation published in [27] is mostly the work of my co-author, Miklós Kálmán. Besides the compression, our method can also be used for semantic XML validation, which was also outlined in [27].

Chapter 2

Size optimization with ARM Code Compression

Traditional embedded systems usually have a limited storage capacity. One of the simplest solutions for saving space is that of compression. One of the nicest ways of compression is that if the file system itself has a transparent compression feature, then neither the users nor the developers have to deal with it.

The compressing file systems usually use a general purpose compressor. Its compression ratio can be improved if we implement more apriori knowledge in the compressor. In the case of embedded systems, a certain amount of space is used to store binary code. As one of the the most frequent architectures available is ARM, we decided to develop an ARM code compressor.

There are numerous compression methods presented in the literature, many of which are overviewed in a survey by Árpád Beszédes et al. [5]. These methods usually can be divided into a model part and a coder part. Our method is also like this. Based on previous studies by C. W. Fraser [13] and M. Garofalakis et al. [14], we used special decision tree(s) as the model and an arithmetic coder [44] as the coder.

The implementation of our algorithm is called ARMlib. It was tested on real iPAQ handheld machines using JFFS2 file system. JFFS2 [45] was especially designed for flash devices and it uses a general purpose compressor called zlib (the compressor library of gzip) for transparent compression. We improved the compression capabilities of JFFS2 to be able to use both ARMlib and zlib compressors. On a file system image with size 25MB uncompressed, 1.7 - 2.6MB was saved based on the parameters of the algorithm. This means that the compressed image was 12.6 - 19.4% smaller when ARMlib was used than the original (zlib-only) compressed image; hence this amount of flash memory could be saved.

The drawback of ARMlib is its speed. The boot time of the iPAQ when ARMlib

was used was 2-4 times the original boot time (when only zlib was used). However, the slower performance is not a nuisance for a regular user in general because users usually do not switch off the mobile devices, but only turn on the sleep or suspend mode; and the Linux kernel also uses a cache that is optimized for speeding up the regular use.

The main contributions of this chapter are the following. First, we introduce a decision tree based code compression method which combines Fraser's idea of using decision trees as compression models [13] and the effective tree construction algorithm of Garofalakis et al. [14], which was originally intended for building trees for classification. We applied this method in a real environment [18]; namely the JFFS2 file system in the Familiar Linux distribution was modified to use both zlib and our method on an iPAQ machine.

Our compression framework is now a the part of the official JFFS2 and Linux kernel. The compression algorithm itself is patented with US patent number 6,917,315 [40] under the name of the authors.

2.1 Background

In this section, we will briefly overview what compression means and how it usually works, then we will briefly describe what decision trees are and do.

2.1.1 Compression

The term "compression" here is defined as in [5]: "storing data in a format that requires less space than usual". In other words: represent some data in a form that is smaller than the original representation. The term "decompression" is the inverse of compression, thus restoring the data to its original form.

The theory behind compression is based on results of *information theory*. Below, we will review some terms used in information theory.

We will define the input of a compression method as a sequence of input symbols. These symbols may be the bits or bytes of the input as well as more complex entities. These entities are usually called tokens. The input sequence may contain values from a fixed set of symbols (token values). The basic idea behind most compression algorithms is to assign a *code* to each symbol in such a way that the sequence of the codes will be shorter than the sequence of the symbols.

Each symbol in the input has a *probability* value. By assigning a shorter code to a more frequent symbol and a longer code to a less frequent one, the overall size of the output sequence will be smaller than the original size of the input sequence. Most compression methods utilizes this idea to produce a smaller output sequence.

Formally: Let $A = \{a_1, a_2, ..., a_N\}$ be a set of symbols. Let $X = x_1, x_2, ..., x_m$ be a sequence with $x_i \in A, (i = 1..m)$. Now, all $a \in A$ has a $P_X(a) \geq 0$ probability in the sequence X, with $\sum_{a \in A} P_X(a) = 1$.

Now, we can compute the *information content* of X. The less information is stored in X, the shorter the encoded sequence can be used to represent it. The information content can be measured by the entropy of X using the following formula:

$$H_A(X) = -\sum_{a \in A} P_X(a) \log_2 P_X(a)$$

This formula gives the minimum average number of bits required to encode one symbol in the sequence X. If $H_A(X)$ is multiplied by m, we get the theoretical minimum size (in bits) of the encoded sequence of X.

2.1.2 Compression model

The compression method can be separated into two parts: (1) gathering information about the input sequence, and (2) assigning suitable codes to the tokens. The first component is called the *modeller*, while the second component is called the *coder*. The latter uses the model provided by the modeller during the code assigning process. Most compression methods contain a separate modeller and coder, although these are usually fine-tuned together. Many modellers can be applied with the same coder and vice versa, so these two can be treated as separate topics of research.

The goal of the model is to provide a good probability distribution on the next token in the input: for each token value say what the chance of the event that the next token has the said value is. Modelling can mean almost anything: the simplest model is the probability distribution of the token values in the input, but the probability distribution provided by the model may vary from token to token. For example, the values provided by the model may depend on the value of the last token.

Coders can also come in different forms. There are coders that directly assign a codeword to each individual input token value (e. g. Huffman) and there are coders that assign a final codeword to a sequence of tokens (e. g. arithmetic). The *decoder* implements the inverse of the coder (it restores the tokens from the codes); however it uses the same model that was used by the coder.

2.1.3 Code compression

Code compression covers the compression of almost any form of a program, including intermediate representation or binary program code but excluding source code (which is rather a special case text compression than code compression). Although

all these forms are considered "code", there may be many differences between them. For instance, the IR code may be some kind of tree (e. g. AST - Abstract Syntax Tree), while the binary code is a sequence of machine instruction.

2.1.4 Decision trees

Decision trees are used for storing information about a set of objects. More precisely, decision trees are trees that provide some information about the target-attribute of an object using some other attributes of it. A typical application is the classification of objects, where the object is placed into one of the given classes based on some properties (attributes) of the object. [37, 38]

In Figure 2.1, the objects are given by their attributes and assigned with a class (+ or −). The decision tree in this figure encodes this classification.

ATT#1	ATT#2	ATT#3	CLASS
1	A	x	+
2	A	x	-
3	B	y	+
3	C	y	+
2	B	x	-
3	B	x	-
1	A	y	+
1	C	y	-
2	A	y	-
1	B	y	-

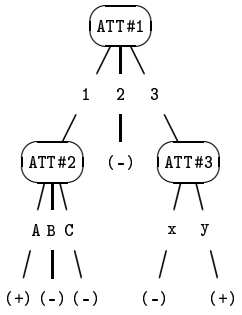


Figure 2.1: A simple decision tree

The tree contains an expression with attributes (*predictors*) in each of its internal nodes. Such an expression may be the attribute itself (as in the example) or a comparison of the attribute with one of its possible values or some more complicated expressions. An outgoing edge is assigned to each of the possible results of the expression in the node. These edges always end in subtrees. In the leaves of the decision tree, information (target-attribute) is stored that corresponds to the decisions made on the path from the root to the actual leaf.

To extract information about an object (which is given by its attribute values) from a decision tree, first evaluate the decision in the root using the attribute values of the given object, then check the end of the edge assigned with the result. If there is a subtree then repeat the process on it. When a leaf is reached it will contain the required information.

Decision trees are mainly used because they can be built automatically. Tree construction algorithms require only a great number of examples (where the target-

attributes are also known) to automatically create a decision tree. The best known decision tree building algorithms are ID3 [37] and C4.5 [38].

2.2 Previous works

Next, previous studies on binary code compression and decision tree building are described, and their advantages and drawbacks are mentioned.

2.2.1 Code compression methods

Code compression is used for a variety of reasons. The goal may be the reduction in energy consumption, which may result in a smaller stored size and/or the smaller data size sent through the channels between hardware elements. Sometimes compression has no other reason than that of saving space, thus improving the capacity of the storage device.

Hence, we concentrate on binary code compression although we will borrow some concepts from other code compression techniques as well.

Benini et al. created a *transparent* compression method [4], where the compressed code is decoded by a hardware unit between the memory and the CPU. The decoder unit is a simple table that contains the most frequent 255 codes, assigned to a byte-long codeword. The 256^{th} value is an escape character to state an uncoded instruction. The most frequent codes come from statistics and the codewords are assigned using the Minimal Hamming Distance.

Wolfe et al. described a *block-based* compression method usable on hardware with memory cache [3]. The cache pages are stored in the memory in compressed form and decompressed into the cache when a cache miss occurs. The connection between the in-cache and real memory addresses is resolved using a Line Allocation Table (LAT). Many coders were tested, and a version of Huffman coding was recommended for use by the authors.

Breternitz and Smith enhanced the previous method [8] and eliminated the LAT. The memory addresses in the code was modified to contain a cache page address and an offset of the target instruction. This solution created more problems, though, like when the program runs through a cache page boundary without a jump. (This was later solved by automatically generated jump instructions.) The best compression ratio they achieved was 0.56.

Laketsas et al. improved Wolfe's method [30]. They proposed the decomposition of RISC instructions into *streams*, thus the different parts of the instructions (e. g. operation code, target register) are encoded in different sequences. They also proposed arithmetic coding. Their models were Markov models and dictionaries. Their average compression ratios were $0.5 - 0.7$.

The work of Lefturgy et al. is similar to Wolfe's solution [29], but they assign codewords not just to single instructions, but to instruction sequences too. The model used is a dictionary and the codewords have a fixed size. The CodePack method of IBM [22] is similar to this, but is much more complicated. The 32 bit-long instructions are divided into two 16 bit-long parts and then these are encoded with a variable-length coding. The decoder is also a hardware unit, but the software-based decoding was also investigated. The compression ratios for these methods were about 0.6.

The methods above are designed for hardware decompression. Their main task was to reduce the energy consumption of the embedded systems, hence the simplicity was more important than the greater compression ratio. Their models are usually based on the use of some kind of a dictionary. In our case, the energy consumption is secondary, the decompression can be performed by software and the compression ratio is more important, so our model can be more complex.

In [13], Fraser created a model for code compression using machine learning methods. The coder and decoder are not the subject of his paper; he focused on finding the relevant statistical information existing in the code to be compressed and automatically extracting it.

Fraser worked on an intermediate representation (*IR*) not on binary code. The information he extracted from this representation are probability distributions usable by his coder. He stored these distributions in the "leaves" of this decision tree-like model. To do this, predictors are required that describe the context of the actual token. Fraser used the last 10-20 token values before the actual token (so-called "Markov" predictors) as predictors, and some computed predictors like stack depth. He utilized a simple tree building algorithm to infer his model automatically. It gets a large number of internal representation code and builds the tree for this data set. Fraser reduced the size of the model making a DAG (*directed acyclic graph*) from the tree by merging similar leaves. The results gave 0.19 compression ratio on IR code. But these results did not include the size of the model, which can be very large.

2.2.2 Decision tree building

Decision trees can be automatically generated on large and representative *training data sets* and hence are easy to use. One of the best known tree building algorithms is ID3 [37]. It is based on entropy gain. Let X be a set of objects, T is the target-attribute, A is a non-target attribute. Let $\mathbf{T}$ and $\mathbf{A}$ denote the set of target and non-target attribute values. Now define the probability $P_X(t)$ and set $X_{A,a}$ for all $t \in \mathbf{T}$ and $a \in \mathbf{A}$ as

$$P_X(t) = \frac{|\{x|x \in X, T(x) = t\}|}{|X|},$$

$$X_{A,a} = \{x | x \in X, A(x) = a\}.$$

Now the entropy gain of attribute A on the set of objects X using the formula $H_S(X)$ introduced in Section 2.1.1 is

$$EG_A(X) = |X| H_{\mathbf{T}}(X) - \sum_{a \in \mathbf{A}} (|X_{A,a}| \, H_{\mathbf{T}} (X_{A,a})).$$

Tree building works as follows: a set of objects (X) whose attributes are known is assigned with the root; the *best attribute* (A) which produces the highest entropy gain ($EG_A(X)$) is then selected; the set of objects is split into subsets based on the best attribute ($X_{A,a}$) and each subset is assigned with a child of the root. The algorithm is then invoked for the children. If each attribute produces a negative entropy gain, the node becomes a leaf encoding information of the target attribute values of the objects.

One typical problem can be that of *overfitting*. If a tree trained on a noisy training set (that contains many errors) exactly fits the training set, then the value the tree gives will also contain errors. Two kinds of methods can solve this problem.

The first is applied during tree building. When a new node is examined whether to be expanded or not, some *stopping criteria* is checked. These may depend on various properties of the node and the assigned training set. If any of these criteria becomes true, the node is not expanded.

The other method is applied on the fully built tree. The subtrees are examined and replaced with leaves if necessary. This is called *pruning*. It gives a more accurate result than the previous method because all subtrees are well known during pruning, while in the previous method the subtrees are not known before the stopping decision. But it may happen that a big subtree is built first and then dropped during pruning.

An enhancement of ID3 algorithm is C4.5 [38]. It contains a pruning algorithm that works on a rule-set derived from the tree. Each leave is represented by the decisions made in the tree from the root to that leaf, then these rules are merged and sorted, and the resulting rule list is used.

In [14] Garofalakis et al. introduced some methods for the efficent building of decision trees. Their trees are built for classifications, and used the MDL (Minimal Description Length) measure for pruning.

Their pruning method works on the tree. Let R be the root of the (sub)tree to be pruned and C_{leaf} be the MDL cost of a leaf that encodes the information of the training set assigned with R. The C_{node} cost of the (sub)tree can be recursively computed. The pruning algorithm works as follows:

```
Procedure Prune(R)
    C_leaf := cost of a leaf at node R
    C_node := Σ_{i∈Children(R)} Prune(R_i) +
                  cost of encoding internal node R
    If C_leaf ≤ C_node Then
        Replace the subtree rooted at R with a leaf
        Return C_leaf
    Else
        Return C_node
```

In [14], two enhanced versions of this methods were described too. In the first, the size of the tree (number of nodes in it) can be maximized by replacing subtrees of the pruned tree with leaves using a dynamic programming method. In the second, the minimal precision of the tree can be set. This uses the above method after initially setting the bound on the tree size to 1, then increasing the bound by 1 until the precision of the tree reaches the required value. In this article a third method was also described that can use the bounds set for the tree size during the tree building phase (using the *branch and bound* technique).

2.3 ARMlib - ARM Code compression

Here our compression method is described via *ARMlib*, which is a function library that contains functions for ARM binary code compression.

In ARMlib we use a decision tree as a model and an arithmetic coder [44] is used as a coder.

The building of the decision tree is based on the papers by C. W. Fraser [13] and M. Garofalakis et al. [14], which were adapted and optimized for ARM compression on real machines.

The tree building automatically optimizes the model for image size. Not only is the compression ratio minimized (because it would result in a very precise, but huge model), but also the size of the tree itself. These models provide a good performance (i. e. greatest compression) on code sequences that are similar to the code they were trained on.

By setting the parameters of the training, the efficiency and speed of the compression can be varied.

2.3.1 The parts of ARMlib

ARMlib consists of 3 main parts:

model generator module creates the model from a great number of example programs, which forms the training data set.

compressor module compresses binary ARM code using a previously generated model.

decompressor module decompresses the compressed code to binary ARM code using the model.

The tokenizer and detokenizer algorithms are also the part of the ARMlib. Tokenizer transforms the raw ARM binary code into the sequence of tokens usable by the modeller and coder, and detokenizer transforms the sequence of tokens into ARM binary codes after decompression. Figure 2.2 shows how the parts of the ARMlib work together.

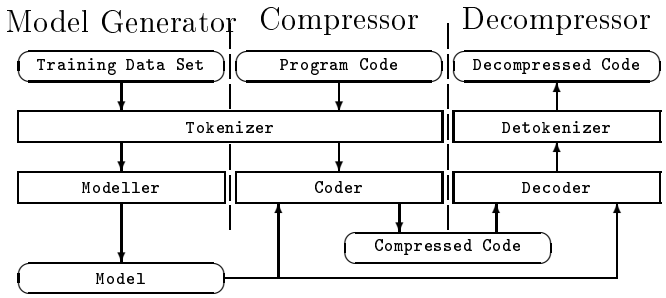


Figure 2.2: The structure of ARMlib

2.3.2 Pre- and post-processing of the ARM code – the tokenizer

The input of model generation, compression and the output of decompression are ARM binary code. The methods applied work on a sequence of tokens, so we need to transform the raw 32 bit-long ARM codes into tokens and the tokens back to ARM codes. These transformations are made by the tokenizer and detokenizer modules.

ARM instructions are 32 bit-long that would be too large for our decision tree building method, so we divided the instructions into more (and smaller) tokens. In ARMlib the instructions are split into 8 tokens, which are each 4 bits long, so the first step of model generation and compression is to create a sequence of tokens from the sequence of ARM instructions, which is performed by the tokenizer module. The last step of the decompression is to create the ARM instruction sequence from the token sequence, which is the task of the detokenizer.

The 4 bit-long tokens were chosen for two reasons. The first one is mentioned above: it is small enough to build smart decision trees. The second reason is that the ARM instructions mostly consist of parts whose length in bits are a multiple of 4. Let us see a typical ARM instruction, and its 4 bit-long components:

COND	INSH	INSL	OP1R	DEST	OP2H	OP2M	OP2L

Here `COND` is the condition parts, `INSH` and `INSL` are the code part, `OP1R` is the first operand (register number), `DEST` is the destination register, `OP2H`, `OP2M` and `OP2L` are the second operands (with many possible meanings) of the instruction. For example, the code of the assembly statement "`ADDLT R2,R4,R9,LSR #3`" is

LT	ADD		R4	R2	LSR #3		R9
1011	0000	1000	0100	0010	0001	1010	1001
COND	INSH	INSL	OP1R	DEST	OP2H	OP2M	OP2L

After taking lots of measurements we decided to reorder the tokens, and use the following order: `INSL`, `INSH`, `OP2M`, `OP2H`, `OP1R`, `DEST`, `OP2L`, `COND`. This helped to achieve a better compression ratio in model generation and compression. For instance, `COND` is much more predictable from `INSL` and `INSH` than `INSL` or `INSH` from `COND`.

The method can be easily applied on any architecture whose instructions have a fixed length. And if the bits of the instructions can be assigned into functional groups, the method might be as successful as in the case of ARM.

2.3.3 The model

ARMlib uses special decision trees as a model.

It has 16 reduced predictors and 1 computed. The reduced ones are the last 16 tokens (2 ARM instructions) before the actual token. The computed one identifies the order of the token in the actual ARM instruction (1-8). These predictors are very simple and fast to compute, which is good from a speed point of view.

ARMlib can use two kinds of internal nodes (similar to ID3 and C4.5 trees):

multivalue decision node, which has as many children as the predictor (8 in the case of the computed predictor, 16 otherwise)

binary decision node, which compares a predictor with one of its possible values. It is a less-than comparison, and the node has two children, namely one for the *true* result and one for the *false* result.

To build the tree we used ID3 with an MDL (Minimal Description Length) measure. The MDL measure used by Garofalakis et al. [14] is not appropriate for building an optimal tree in our case because the size of the compressed code and the size of stored version of the tree have to be minimized together. As computing these values precisely and efficiently was no easy task, we decided to use the following computational approximations:

- The full size of the tree can be computed, but in practice it is also compressed by a general purpose compressor (JFFS2 uses zlib). To estimate the stored size of the tree, we use a simple pre-defined constant compression ratio of the general purpose compressor.
- The size of compressed code can be estimate by the sum of the entropies of the training sets associated with the leaves of the subtree. This is also a good approximation if the output of the coder is close to the entropy.

2.3.4 The coder

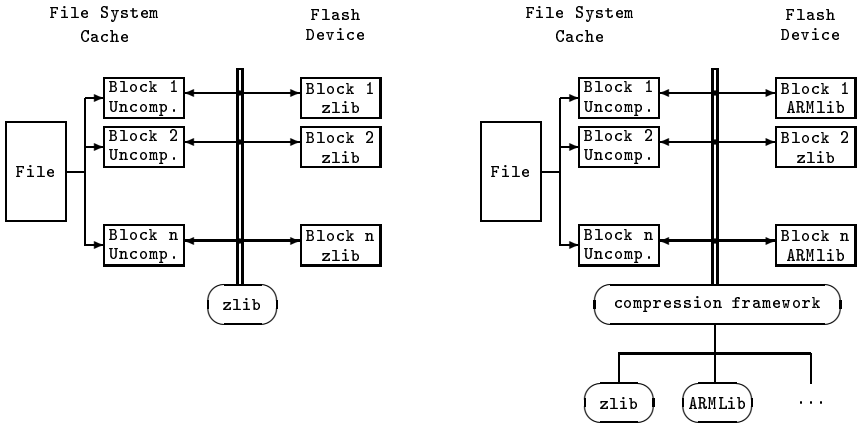
ARMlib uses arithmetic coder [44]. The concept of the arithmetic coder is not to assign codewords, but assigns an interval between 0 and 1 to the tokens. It also assigns an interval to the sequence of tokens (which interval is computed from the intervals of the tokens). The codeword assigned to a sequence of tokens is a random number in the final interval stored with high precision enough for correct decoding.

Arithmetic coder produces a codeword whose length is very close to the entropy. Technically the halving of the size of the actual interval means one more bit in the codeword. This allows the arithmetic coder to virtually assign a fraction of a bit to a token whose probability is greater than 50% (while for example Huffman coder assigns at least 1 bit to each tokens regardless its probability).

2.4 An ARMlib implementation

ARMlib were tested in a real environment, on *iPAQ* handheld machines [23]. These machines have *StrongARM* processors, 32MB of memory and 16MB of flash. The operating system used was *Familiar Linux* with a modified Journaling Flash File System 2 (JFFS2).

JFFS2 stores the files logically in fixed size (4K) blocks, and compresses these blocks in a transparent way when they are stored in the flash memory. It means that the user of the file system may not notice the compression, only that there is more space on the device, and the system slower a little. Originally JFFS2 used only a general purpose compressor called zlib. (See Figure 2.3a)



a: Original **b:** Advanced

Figure 2.3: The improved compression of JFFS2

In our implementation (see Figure 2.3b), we replaced *zlib* by a *compression framework*. The method of the compression depends on the settings of the compression framework. If it is set to optimize for size, then all compressors will be called, and the smallest result will be stored. Each compressed block of the file system stores the information whose method it was encoded by, so decompression can be preformed by the appropriate compressor.

The whole use scenario is the following:

1. Model and file system image generation on the host machine
   (a) Process the file structure of the system to generate the block (4Kb) as the normal file system image generation method.
   (b) Separate the blocks which contains ARM code. They will be the training set of the model generation.
   (c) Build decision tree from the ARM blocks. This process has two parameters:
      Type of the decision tree nodes may be a binary or multi-value tree - as described above.
      The number of the decision trees : it means that the process can generate one or more decision trees with a randomly separated training set. Using more decision trees gives a different performance, as we will see later on.
   (d) Generate the file system image. All blocks are compressed with both ARMlib and zlib, and the smaller variant is kept. The block stores the

identifier of the model it was compressed with, if necessary (if there was more than one model). Model file blocks and the kernel (containing the algorithm of ARMlib) can only be compressed with zlib.

2. Upload the file system image to the device. The bootloader (which only knows zlib) loads the kernel modules including ARMlib. At initialization the ARMlib loads its model file(s) into the memory, and it is ready for use.

3. Use the system: all blocks will decompress with the corresponding compressor on the fly, the blocks of the new data or programs will be compressed by both zlib and ARMlib, and the smaller variant is kept, as in the file system image generation phase.

2.5 Results

The test environment a handheld device namely iPAQ H3600. The modeller and the coder ran on a host machine, but coder and decoder functions were used on the iPAQ. This environment was used to test the practical usability of the method. The uncompressed size of the image of the iPAQ was 25MB.

2.5.1 Size

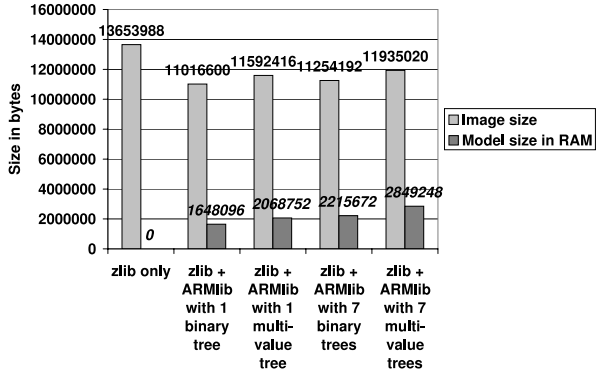


Figure 2.4: Compressed sizes

Different image sizes can be seen in Figure 2.4. The size reductions relative to the image compressed with zlib only lie between 12.6% and 19.3%. Trees with binary nodes only are better than trees with multi-value decisions (because multi-value nodes can be substituted with binary nodes, but not vice versa). A small

increase in image size was observed when the one model was replaced by seven smaller models, which were trained on seven randomly separated distinct parts of the original training set. The results show that one tree is more effective in compression size (at least when the split of the data set is random) than the seven smaller models together, but the (de)compression speed is better for the seven models than for only one.

The same figure shows the sizes of the models in the memory. Binary models are smaller, and the seven smaller models require more memory in total.

2.5.2 Speed

The speed of the compression is worse than zlib's own speed. Compression and decompression in ARMlib requires almost the same amount of time, but this makes decompression very slow relative to zlib (in zlib, decompression is 10 times faster than compression). Fortunately, Linux caches the file system. This means that not all accesses to the block provoke compression or decompression, usually it is just the first and the last accesses. This improves the average speed of the file system, and makes ARMlib's relative slowness almost undetectable to a regular user.

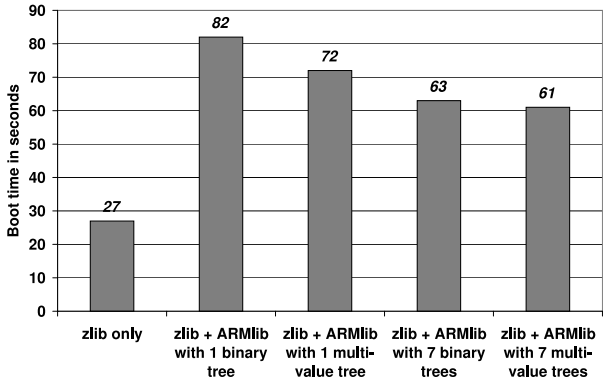


Figure 2.5: Boot times

The boot time is 2.3-3 times that of the original boot time (Figure 2.5), but considering the savings in size (12.6-19.3%) this seems to be a good compromise.

2.6 Summary

Here, our goal was to create a code compression method which compresses ARM binary codes better than current algorithms do. We combined a method that used

decision trees as the compression models [13] and an efficient tree building algorithm [14]. In addition, the tree building and pruning phases were also combined within the method. The method was intended for the ARM instruction set, but it can be applied to other instruction sets that satisfy a fixed set of requirements.

The implementations of the functions of the method were collected in a library called *ARMlib*, which used an arithmetic coder [44]. ARMlib was tested in the JFFS2 file system [45] on iPAQ machines. This modified file system [18] used either zlib or ARMlib compression on a block. The use of ARMlib reduced the image size by 12.6-19.3% depending on the parameters of tree building. The method can be used in situations when size is more important than speed.

The compression framework is now the part of the official Linux kernel. The compression algorithm is patented with US patent number 6,917,315.

The compression algorithm is a common result [15] with my co-author, Tamás Gergely. The structure of the efficient method for integrating the algorithm into the Linux Kernel, and the implementation itself, are my own results. The authors of the corresponding US patent [40] are myself, Gergely Tamás and Árpád Beszédes, and its beneficiary is the Nokia Corporation.

Chapter 3

Performance optimization with an Improved B+ Tree

One of the most important parameters of a multimedia mobile device is its performance. Performance is a complex concept, but it is mostly related to speed, memory consumption and efficiency.

Here, we introduce a new data structure and algorithm designed for flash file systems that work more efficiently than the previous technical solutions. We begin by introducing the workings of flash chips, then describe the previously most prevalent Linux flash file system, namely JFFS2. Its biggest weakness is its indexing method: it stores the index in the RAM memory, not in the flash memory. This causes unnecessary memory consumption and performance penalties. Next, we introduce the new method in a step-by-step fashion, with an efficient combination of storing the index in memory and flash after taking factors like data security and performance into account.

This new solution was implemented in the UBIFS file system by our department in cooperation with Nokia, and it is now an official part of the Linux kernel.

3.1 How the flash memory works

Flash memory [6] is a non-volatile computer memory that can be electrically erased and reprogrammed. One of the biggest limitations of flash memory is that although it can be read or programmed one byte or word at a time in a random-access fashion, it must be erased one "block" at a time. The typical size of a "block" is 8-256KB.

The two common types of flashes are NOR and NAND flash memories, whose basic properties are summarized in Table 3.1 [33].

One of the drawbacks of the flash system is that its erase block can be erased

	NOR	NAND
Read/write size	Can read/write bytes individually	Can read/write only pages (page size can be 512 or 2048 bytes)
I/O speed	Slow write, fast read	Fast write, fast read
Erase	Very slow	Fast
XIP (Execute in Place)	Yes	No
Fault tolerance, detection	No	Yes
Price/size	Relatively expensive	Relatively cheap

Table 3.1: Difference between NOR and NAND flash

about 100,000 times, and afterwards the chip will be unstable. This is why most of the ordinary file systems (FAT, ext2/3, NTFS, etc.) are unusable on flash directly, because all of them have areas that are rarely rewritten (FAT, super block, etc.), and this area would soon be corrupted.

One of the most common solutions to balance the burden of the erase blocks is FTL (Flash Translation Layer) [24], which hides the physical erase blocks behind a layer. This layer uses a map to store data about what the corresponding physical erase block is for each logical number. Initially this map is identical, so for example logical block 5 is mapped to physical block 5. This layer also contains an erase counter for each block (how many times it was erased). If this counter reaches a high number (relative to the average), the system will exchange two erase blocks (using the map), selecting an erase block which has a relatively low strain. This method is used in most pen drives to keep the burden low. It works quite well in practice, but does not provide the optimal solution to performance problems. For instance, to overwrite just a few bytes (such as a pointer in a search tree), an entire erase block ($\sim$128KB) has to been erased and reprogrammed.

Accordingly, especially in the case of root file systems, it is worthwhile using flash file systems which are designed specifically for flash devices. Now we will discuss Linux flash files systems (JFFS and JFFS2), which are freely available to everyone.

3.2 JFFS, JFFS2: flash file systems without flash index

The basic idea behind JFFS [45] is quite simple: the file system is just a concentric journal. In essence, all of the modifications on the file system are stored as a journal entry (node). When mounting, the system scans this journal and then replays the events in the memory, creating an index to register which file is where. If the journal entries are such that the device is nearly full, the system performs the following steps:

- From the beginning of the used area copy all of the journal entries which are still valid, so the corresponding file is not deleted or overwritten.
- Erase the emptied area (erase block), so the there will be new space to store the new journal entries.

This very simple approach eases the burden on the erase blocks, but it also has its drawbacks:

1. If a dirty (deleted or overwritten) data area is in the middle of the current journal area, to free it, it is necessary to copy half of the entire journal.
2. When mounting, it is necessary to scan the entire medium.

Problem 1 is solved by the new version of this file system called JFFS2. It utilizes lists to register the dirty ratio of erase blocks, and if free space is required, it frees the dirty erase blocks (after moving the valid nodes to another place). There is a slight chance that it will free clean erase blocks as well, just to ease the burden, but this will rarely occur.

Problem 2 is only partially solved by JFFS2. It collects information via erase blocks needed when mounting, and it stores them at the end of the erase blocks, so only this needs to be scanned. Afterwards it attempts to minimize the size of the index in the RAM. However, the memory consumption and the mounting time are still linearly proportional to the size of the flash, and in practice over 512MB may be unusable, especially in the case of large files – e.g. video films.

Because the root of this problem lies in the base data structures and operating method of the JFFS2, we should construct a new file system to eliminate the linear dependency. To achieve this, it is necessary to store index information on the flash so as to avoid always having to rebuild it when mounting.

3.3 B+ tree

Most file systems employ search-trees to index the stored data, and the B+ tree [11] is a special search-tree with the following features:

- It stores records: $r = (k, d)$; k = key, d = data. The key is unique.
- Data is stored only in leaves, inner-nodes are only index-nodes.
- In an index-node there are x keys, and also $x + 1$ pointers, each pointing to the corresponding subtree.
- The B+ tree has one main parameter, namely its order. If the order of a B+ is d, then for each index node there is a minimum of d keys, and a maximum of $2d$ keys, so there are a minimum of $d + 1$ pointers, and a maximum of $2d + 1$ pointers in the node.
- From the above, if a B+ tree stores n nodes, its height must not be greater than $log_d(n) + 1$. The total cost of insertion and deletion is $O(log_d(n))$.

This data structure is used by some database systems like PostgreSQL and MySQL, and file systems like ReiserFS, XFS, JF2 and NTFS. These file systems are based on the behaviour of real hard disks; that is, each block can be overwritten an unlimited number of times. Thus if there is a node insertion, only an update of the corresponding index node is needed at that location. Unfortunately it does not work well with flash storage devices, so it was found necessary to improve the flash-optimized version of the B+ tree.

3.4 Wandering tree

A modified version of the B+ tree can be found in the LogFS file system [25], which is a flash file system for Linux. It is still in the development phase, and probably will be never finished because UBIFS offers a much better alternative. This B+ variant is called a wandering tree. The general workings of this tree can be seen in Figure 3.1.

Like the ordinary B+ tree algorithm, during a node insertion it is normally necessary to modify a pointer at just one index node. In the case of flash memory the modification is costly, so this wandering algorithm writes out a new node instead of modifying the old one. If there is a new node, it is necessary to modify its parent as well, up to the root of the tree. It means that one node insertion (not counting the miscellaneous balancing) requires h new nodes, where h is the height of the tree. It also generates h dirty (obsolete) nodes, as well. Because h

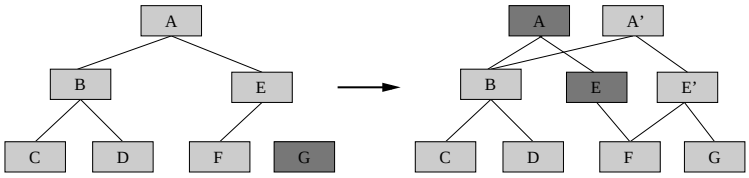


Figure 3.1: The wandering tree before and after insertion

is $O(log_d(n))$, where n is the tree node number, the cost of this operand is still $O(log_d(n))$.

3.5 The TNC: an improved wandering tree

The above wandering tree algorithm still has performance issues because its insert method is inefficient: it requires $log_d(n)$ new nodes, and it also generates a number of garbage nodes.

To solve these problems we decided to improve this algorithm, which now works in the UBIFS file system [1]. Due to its efficient caching technique it is able to collect node insertions and deletions in the memory, so fewer flash operations are required. At the same time, the method can ensure that if there is a sudden power loss (in the case of an embedded system this could happen at any time), there will be no data loss.

This new data structure and the algorithm are both called the TNC (Tree Node Cache). It is a B+ tree, which is partly in the flash memory, and partly in the memory (see Figure 3.2). Its operators are improved versions of those used in the wandering tree algorithm.

3.5.1 Data structure of the TNC

When TNC is not in use (e.g. the file system is not mounted), all the data is stored in the flash memory, in the ordinary B+ tree format.

When in use, some index nodes of the tree are loaded into the memory. The caching works in such a way that the following statement is always true : if an index node is in the RAM memory, its children may also be in the memory, or in the flash memory. But if the index node is not in the memory, all of its children are in the flash memory.

If an index node is in the memory, the following items are stored in it:

- **Flag clean:** it tells us whether it has been modified or not.

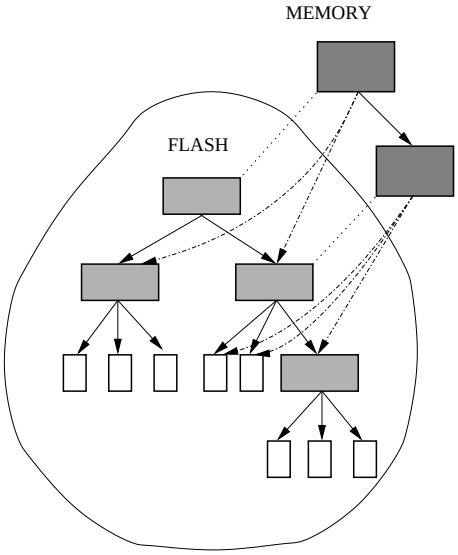


Figure 3.2: The TNC data structure

- **The address of the flash area where the node was read from.** (In the case of being eliminated from the memory, and it was not modified, this address will be registered in its parent – in the memory. If it was modified, a new node will be written out, and the old location will be marked as garbage.)
- **Pointers to its children.** Each pointer stores information about whether the child is on the flash or has been read into memory.

3.5.2 The TNC operations

Using the data structures above, the following operators can be defined:

Search (read):

1. Read the root node of the tree into the memory, then point to it using the pointer p.
2. If p is the desired node, return with the value of p.
3. Find at node p the corresponding child (sub tree), where the desired node is.
4. If the child obtained is in the memory, set pointer p to it, and jump to point 2.

5. The child is in the flash memory, so read this into memory. Mark this child in p as a memory node.

6. Set pointer p to this child, and jump to point 2.

Clean-cache clean-up (e.g. in the case of low memory):

1. Look for an index-node in the memory which has not yet been modified, and for which all of its children are in the flash memory. If there is no such index-node, then exit.

2. Set the pointers in the identified node's parent to the original flash address of the node, and free it in the memory.

3. Jump to point 1, if more memory clean-up is needed.

Insert (write):

1. Write out the data as a leaf node immediately. UBIFS writes them out to the BUD area[1], which is specially reserved for leaf nodes, just to make it easier to recover when necessary.

2. Read (search) all of the nodes into memory that need to be modified using the B+ algorithm. (In most cases it is just one index node.)

3. Apply the B+ tree modifications in the memory.

4. Mark all modified nodes as dirty.

In the method described above node insertions can be collected, and we can apply them together with significantly lower flash overheads.

Commit (Dirty-cache clean-up):

1. Look for a dirty index node that has no dirty child. If found, call it node n.

2. Write out a new node n onto the flash, including its children's flash addresses.

3. Mark the place dirty where the node n was previously located, and update the flash pointer in the memory representation of the node to the new flash address.

[1]There are two kinds of data erase block in UBIFS, namely the BUD erase block and the non-BUD erase block. UBIFS stores only the leaf nodes in the BUD erase blocks, while all other types of data nodes are stored in non-BUD erase blocks.

4. Mark the parent of node n as dirty (if it is not the root node), and mark node n as clean.

5. Jump to point 1 until there is a dirty node.

Deletion:

1. Read (search) all of the nodes into memory that need to be modified using the B+ algorithm. (In most cases it is just one index node.)

2. Apply the B+ tree modifications in the memory.

3. Mark all modified nodes as dirty.

3.6 Power loss handling in the TNC

In the case of power loss, the information stored in the memory is lost. To prevent this from happening, UBIFS combines TNC with a journal, where the following information is stored:

- A journal entry with a pointer to new BUD erase blocks. BUD erase blocks in UBIFS are reserved areas for leaf nodes. If the BUD area is full, a new free erase block will be reserved for this purpose.

- Delete an entry after each node deletion.

- A journal entry after each commit with a list of still active BUD areas.

In the event of power loss, the correct TNC tree can be recovered by performing the following steps:

1. Start with the tree stored on flash.

2. Look for the last commit entry in the journal. All of the events that occurred from that point have to be scanned.

3. All of the node insertions stored in the BUD areas marked in the journal, and all of the deletion nodes stored in the journal have to be replayed in the memory.

3.7 Experiments

Measuring file system performance objectively is not a simple task, because it depends on many factors like the architecture behaviour and caching of the operating system. To avoid these strong dependencies, we decided to measure just the most important factor of the flash file system performance, namely the size of flash I/O operands to evaluate different TNC configurations and examine their properties using the UBIFS implementation.

The method applied was the following: we unpacked the source of Linux kernel version 2.6.31.4 onto a clean 512MB file system, and deleted data using the commands below. During the test, the system counted how many flash operands (in terms of node size) were made with and without TNC.

```
mount /mnt/flash
mkdir /mnt/flash/linux1
tar xfz linux-2.6.31.4.tar.gz /mnt/flash/linux1
rm -rf /mnt/flash/linux1
umount /mnt/flash
```

We measured the performance using different TNC configurations. A TNC configuration has the following parameters:

TNC buffer size : The maximum size of the memory buffer that the TNC uses to cache. If it is full, it calls commit and shrink operands.

Shrink ratio : In the case of shrink, the shrink operand will be called until this percentage of the TNC nodes is freed.

Fanout : B+ tree fanout number: the maximum number of children of a tree node. ($2d$, where d is the order of the B+ tree.)

Table 3.2 and Figure 3.3 show the results of measuring the flash performance when the TNC *buffer size* and *shrink ratio* were varied. As can be seen, the TNC saves 98.2-99.4% of the flash operands. Increasing the TNC size, more of the flash operations are saved, but varying the shrink ratio has no noticeable effect here.

Table 3.3 shows what happens if we change the fanout value of the tree. The number of TNC nodes decreases, but the size of a TNC node increases because a TNC node contains more pointers and keys. The size of the flash operations is the product of these two factors, and it has a minimum fanout value of 32.

In the remaining tests we took different samples from the source code of Linux kernel version 2.6.31.4. Table 3.4 and Figure 3.4 tell us the maximal TNC size (setting no limit) when the fanout is varied, and the size of the I/O operands (size of the "file-set" above) as well.

Max. TNC size	Without TNC	With TNC	Shrink Ratio	With TNC / without TNC
5000	2161091	38298	25 %	1.77 %
10000	2211627	31623	25 %	1.43 %
15000	2191395	24632	25 %	1.12 %
20000	2244013	20010	25 %	0.89 %
25000	2192044	12492	25 %	0.57 %
5000	2163769	36273	50 %	1.68 %
10000	2250872	31570	50 %	1.40 %
15000	2225334	22583	50 %	1.01 %
20000	2225334	20002	50 %	0.92 %
25000	2183596	12457	50 %	0.57 %
5000	2215993	36759	75 %	1.66 %
10000	2290769	32578	75 %	1.42 %
15000	2244385	29956	75 %	1.33 %
20000	2238633	20002	75 %	0.89 %
25000	2205709	12958	75 %	0.59 %

Table 3.2: The number of the flash operations (measured in terms of node size)

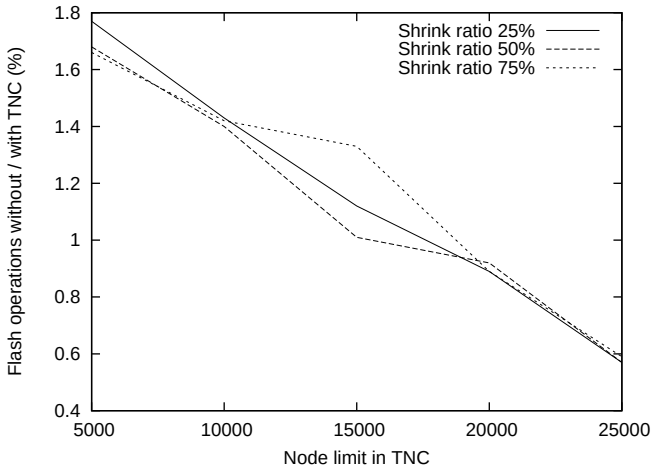


Figure 3.3: The performance of TNC flash operations compared to those got using the simple wandering algorithm

It is a great help to know how it behaves, especially if we want to use this technique in an embedded system where the system performance and the maximal

Fanout	Without TNC in nodes	With TNC in nodes	Max TNC in nodes	TNC node size	Max TNC in MB	Flash ops in MB
4	1134784	48392	64801	176	10.88	8.12
8	2168308	12405	23189	304	6.72	3.6
16	1304212	3577	9662	560	5.16	1.91
32	1024363	1317	4669	1072	4.77	1.35
64	1140118	3420	3671	2096	7.34	2.35
128	767005	1245	1586	4144	6.27	3.35
256	930236	1641	980	8240	7.7	4.35

Table 3.3: The effect of varying the TNC fanout

I/O Size (MB) \ Fanout	8	16	32	64
50	3302	1456	703	351
100	6364	2818	1355	671
200	12925	4620	2224	1106
400	23518	8978	4282	2861
600	43320	18426	8846	5840
800	44948	22070	12273	8527

Table 3.4: The maximal TNC size as a function of tree fanout

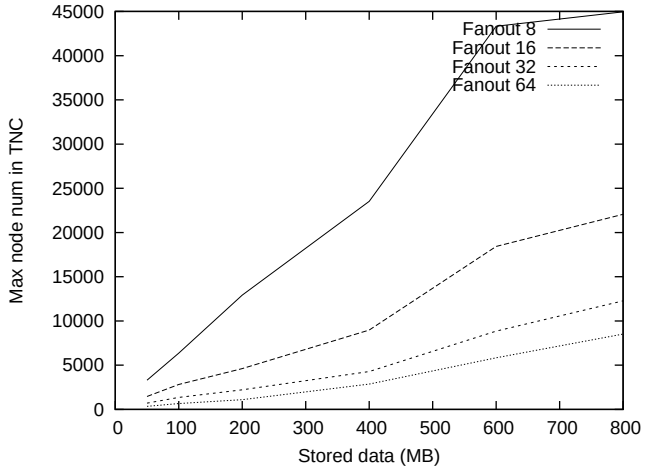


Figure 3.4: The maximal TNC size as a function of tree fanout

memory use of the file system are both of crucial importance.

3.8 Related Work

The authors of [35] outlined a method that had a similar goal to ours, namely to optimize the B+ tree update on a flash drive. The method collects all the changes in the memory (in LUP = lazy-update-pool), and after it has filled up, data nodes are written out in groups. It also saves flash operations, but unlike our method, using LUP means a lower read speed because, before searching in the tree, it always has to scan the LUP. In the case of TNC, there is usually a higher read speed because the nodes (at least the modified ones) are in the memory. Our method is power-loss safe, but the authors of [1] do not discuss what happens when the information is stored in the LUP. The advantage of their method is the following: the node modifications can be grouped more freely (not just sequentially), so it may be easier (and require less memory) to close the tree operations intersecting the same tree area.

The goal outlined in [46] is also a B+ tree optimization on a flash memory. It collects as well any changes made in the memory and stores them in the "Reservation Buffer". It is filled up and these changes are written out and grouped by a "Commit Policy" into flash as an "Index Unit". It makes use of another data structure called the "Node Translation Table" to describe which node has to be transformed by which Index Unit. To search in the tree, it is necessary to scan both the Node Transaction Table and the Index Units.

The method described in [21] is essentially an improved version of that described in [46]. Instead of the simple "Reservation buffer", it utilizes the "Index Buffer", which monitors the tree modifications and if any intersect the same node, it closes them or, where possible, deletes them. In the case of commit, it collects data concerning the units belonging to the same nodes, and writes them out to one page.

3.9 Summary

Here, the author sought to improve the wandering tree algorithm used by flash file systems so as to make it more efficient and save over 98% of the flash operands. It has a power-loss safe variant, and it has a much better performance than a simple wandering tree.

The new generation of the Linux flash file system (UBIFS) uses this algorithm and data structure, and allows one to use a flash file system efficiently on 512MB or larger flash chips. This implementation is now part of the Linux kernel mainline,

and is used in the Nokia N900 smart phone.

These results are my own results, and were published in [19].

Summary in English

Cell phones, DVD players, MP3 players, GPS receivers, car electronics and similar devices nowadays have an important role in our lives. One of the main features of these devices is that their resources are generally much more limited than the resources of the personal computers. So optimizing the software components is always a priority.

In this thesis, three corresponding results are presented, namely one XML-related result and two flash file system optimizations.

1. XML semantics extension and compaction

The XML (eXtensible Markup Language) is one of the most widely used structured file formats. With a practical application of our results, XML files can be stored in a more compact form, which can especially be useful in embedded systems and network applications.

The main idea behind our results is based on an analogy between XML documents and attribute grammars, namely the pair of the AG non-terminal symbols in XML is XML elements, while the rules of the AG formal grammars are related to the XML element specifications in DTD/XSD. Attributes can also be defined in both environments, but there is no way one can define semantic rules in an XML environment. To bridge this gap, we defined a new metalanguage called SRML, where semantic computation rules can be defined for XML attributes.

Two basic operations were defined in our method. The *reduce* operation removes the attribute occurrences from the XML file, which are calculated using the rules defined in SRML. It then produces a *compacted* version of the XML file. The task of the *complete* operation is just the opposite. Semantic functions of SRML can come from two sources: it may be produced by an expert, or by applying machine learning algorithms. The two methods can be also combined so that the learning algorithm can use the rules produced by an expert as a starting point.

This compaction technique can also be used to improve XML file compression since a compressor algorithm will be more effective if more correlations can be recognized in the compressed files. Owing to this, an XML compressor (such as

XMill) can generally achieve better compression ratios on compressing XML files than a general purpose compressor like gzip. However, even XML-specific compressors cannot detect the semantic relationships among XML attributes. Hence if we compact an XML file before the compression, it will produce better results compared to compressing it with an XML compressor alone.

Our procedure was tested on CPPML files that were intended to store C++ source files in an XML format. The compaction rate we achieved was between 57-79%. We later improved the efficiency of the compression of XMill XML compressor program by 9-26%.

The results are presented in [20], [26], and [27].

2. Size Optimization with ARM Code Compression

One of the key limited resources of embedded systems is that of storage. One of the simplest solutions for saving space is that of compression. One of the nicest ways of compression is transparent compression integrated in the file system. This is because neither the user nor the application developer need worry about the compression and the decompression itself. JFFS2 has this kind of compression capability, which is one of the most commonly used Linux flash file systems. JFFS2 uses zlib as a compressor, which is a general purpose compressor.

In the case of embedded systems one of the biggest large-scale data types is that of executable code. One of the most popular embedded architectures is ARM. Due to this, our study focused on developing an ARM code compressor that could also be combined with currently available solutions.

Our algorithm is a model-based compression that uses a decision tree model to predict the probability distribution of the next token. This probability distribution is used by an arithmetic coder for encoding and decoding.

Our algorithm builds a decision tree, improving its method to achieve a better compression ratio for ARM codes: the tokens were chosen by knowing the structure of ARM instructions; during the tree building we used MDL-based heuristics as the stopping criteria and we also considered combining it with a general purpose compressor; we investigated different kinds of tree building to get a suitable speed-size ratio for the system.

Besides the development of the new code compression algorithm, we also devised the structure of an efficient implementation. This solution replaces the original zlib compressor in JFFS2 with a compression framework that can be configured to choose the best compression ratio: it calls all the available compressors (zlib and our ARM code compressor) and chooses the smallest result in the case of each block.

The results got using this combined solution meant that a better ratio of 12.6-19.3% could be achieved than that was the original zlib-only solution. The drawback of using this method is speed reduction. In practice, where it is the most noticeable by the user, is the boot time: it is increased from 27 seconds to 61-82 seconds.

Our results were published in [15]. The implemented framework became an official part of the JFFS2 file system and the Linux kernel. The compression algorithm is patented with US patent number 6,917,315 [40].

3. Performance Optimization with Improved B+ Tree

The majority of embedded systems use a flash chip as a storage device because of its low power consumption and robustness (it has no moving parts). For these applications, where efficient data storage is necessary with a read-write mode, file systems are used to store data.

Applying traditional file systems on flash devices is not a straightforward process because most traditional file systems are designed for hard drives, and flash chips have different properties. Because of this, traditional file systems are not usable directly on flash chips, except with the help of bridging middleware (e.g. FTL); but it carries a performance penalty.

A more efficient solution could be that of file systems which are specially designed for flash drives. One of the most popular Linux-based flash file system was the second version of JFFS called JFFS2. This file system, as intended, only stores the index information in the memory, which is built at mount time by scanning the entire flash device. If the flash size is over 512MB, JFFS2 is practically unusable, due to its slowness and high memory consumption.

Because the problem in JFFS2 was so fundamental, it was necessary to design a new file system. Like many file systems, we also decided to use a B+ tree to store index information. In LogFS, the original B+ tree algorithm was modified to handle flash chips, and we improved it so as to produce a really usable solution (called the TNC) that the following properties:

- Its data structure and algorithm handle the data stored on the flash and in the memory as well.
- To achieve a better performance, the method merges tree operations and also does caching.
- Its memory consumption can be restricted, and adapted to the actual amount of the free memory.

- It tolerates power loss very well. This is very important in the case of battery-based devices.

Using the TNC, 98.2-99.4% of the flash operations can be saved compared to that using the B+ tree algorithm of LogFS.

Our results were published in [19]. The TNC has become an official part of UBIFS and the Linux kernel, and it was used in the Nokia N900 smartphone.

Magyar nyelvű összefoglaló

A beágyazott rendszerek életünk egyre több területén körülvesznek bennünket mobiltelefonok, DVD lejátszók, mp3 lejátszók, GPS vevők, autó elektronikai és épületgépészeti vezérlő eszközök, háztartási eszközök egész sora formájában. Technikai szempontból az egyik fő sajátosságuk, hogy erőforrásaik általában sokkal szűkebbek, mint egy személyi számítógépé, így az optimalizáció a rájuk fejlesztett szoftverek esetében kiemelt fontosságú.

A dolgozat három beágyazott rendszerekkel kapcsolatos eredményemet mutatja be, egy XML-lel és két flash fájlrendszerrel kapcsolatosat.

1. XML szemantikus kiterjesztés és tömörítés

Az XML (eXtensible Markup Language) az egyik legelterjedtebb strukturált szöveges fájlformátum. A dolgozatban leírt eredmények egyik gyakorlati alkalmazásával az XML fájlok kisebb helyen tárolhatóak el, ami különösen a beágyazott rendszerek és a hálózati alkalmazások területén hasznos.

Az eredmény alapgondolata az XML dokumentumok és az attribútumnyelvtanok analógiáján alapul: ami az attribútum nyelvtanoknál a nemterminális szimbólum, az XML környezetben az "element", a formális nyelvtan szabályainak pedig a DTD/XSD-ben lévő element specifikáció felel meg. Attribútumok ugyancsak definiálhatók mindkét környezetben, viszont azok szemantikus kiszámításának leírására alkalmas szemantikus függvénynek XML környezetben nincsen megfelelője. Ezt a hiányt töltöttük be azzal, hogy megalkottuk az SRML-nek elnevezett metanyelvet, amellyel definiálhatók az attribútumok kiszámítási módjai. DTD környezetben az SRML külön XML alapú fájlban tárolható, XSD környezetben pedig magába az XSD-be is beleilleszthető, annak appinfo részébe.

Módszerünkben két alapműveletet definiáltunk: a *reduce* művelet az XML fájlból eltávolítja azokat az attribútumokat, amelyek az SRML szabályai alapján helyesen kiszámíthatóak, így az XML dokumentum *kompaktált* változatát képes előállítani, míg a *complete* művelet ennek a fordítottja. Az SRML-ben lévő szemantikus függvényekben leírt összefüggések két forrásból származhatnak: vagy az

adott fájl formátumát ismerő szakértők állítják elő, vagy gépi tanulási algoritmusokkal fedezhetőek fel a nem ismert összefüggések egy-egy adott XML dokumentum vagy dokumentum halmaz esetében. A két módszer kombinálható olyan módon, hogy a tanuló algoritmusnak kiindulásként odaadjuk a szakértői szabályokat, amelyeket az tovább bővíthet.

A kompaktálás használható az XML fájl tömörítésének javítására is, ugyanis a tömörítő programok annál hatékonyabbak, minél több összefüggést tudnak felismerni a tömörítendő fájlokban. Ez az oka annak, hogy a kifejezetten XML-re specializált tömörítőprogramok (mint amilyen az XMill) jobb tömörítési arányt érnek el XML fájlok tömörítésekor, mint az általános célú tömörítők (mint például a gzip). Ugyanakkor az XML specifikus tömörítők sem képesek fölismerni azokat a szemantikus összefüggéseket, amelyek az XML attribútumai között vannak, amire a mi módszerünk viszont képes. Így, ha tömörítés előtt az általunk kifejlesztett módszerrel az XML fájlt kompaktáljuk, majd azt tömörítjük, akkor jobb eredményt fogunk elérni, mintha csak egyszerűen tömörítenénk.

Módszerünket a CPPML fájlformátumon próbáltuk ki, amely C++ programok XML reprezentációjára lett kifejlesztve. Az elért kompaktálási arány 57-79% között mozog. A tömörítés hatékonyságának javítását pedig az XMill XML tömörítőprogrammal próbáltuk ki olyan módon, hogy tömörítés előtt az XML állományt kompaktáltuk. Az így keletkezett tömörített állomány 9-26%-kal kisebb méretű lett, mintha csak tömörítettük volna.

Eredményeinket a [20], [26] és [27]-ben publikáltuk.

2. ARM kód tömörítésen alapuló flash fájlrendszer helykihasználási optimalizáció

A beágyazott rendszerek szűk erőforrásai közül az egyik a háttértár. A háttértár méretének kérdésében az egyik lehetséges megoldás a tömörítés. A tömörítési módok közül is az egyik legkényelmesebb a fájlrendszerbe illesztett transzparens tömörítés, amelynek köszönhetően se a felhasználónak, se az egyes alkalmazások fejlesztőinek nem kell foglalkoznia a be- és kitömörítéssel. Ilyen tömörítési képességgel rendelkezik a JFFS2 is, amely az egyik legszélesebb körben használt Linuxos flash fájlrendszer. A JFFS2 tömörítő algoritmusként a zlib általános célú tömörítőt használja, ahogy a UNIX/Linux alatt rendkívül népszerű gzip tömörítő program is.

Mivel a beágyazott rendszerek esetében az egyik legnagyobb arányú speciális "adatfajta" a futtatható kód, és ezek közül pedig az egyik legnépszerűbb architektúra az ARM, ezért kutatási területként azt céloztuk meg, hogy hogyan lehet hatékony ARM kódtömörítőt fejleszteni, és olyan módon implementálni, hogy az

jól kombinálható legyen a jelenlegi megoldásokkal.

A kidolgozott algoritmusunk egy modell alapú tömörítő, amely modellként döntési fát használ, azért, hogy a következő token valószínűségeloszlását megjósolja. Ezt a valószínűségeloszlást felhasználva aritmetikai kódoló segítésével hajtjuk végre a kódolást/dekódolást.

Algoritmusunk döntési fát épít, amelynek módját az ARM gépikód tömörítésére optimalizáltuk. A tokeneket úgy választottuk meg a ARM utasítások szerkezetének ismeretében, hogy azokból hatékonyan lehessen döntési fa modellt építeni; a modell építését MDL alapú megállási heurisztika alapúvá tettük, és figyelembe vettük annak általános célú tömörítővel való gyakorlati kombinálását; a modell építést többféle módon valósítottuk meg, ami lehetővé teszi azt, hogy az adott rendszerre optimális sebesség/méret arányt kiválaszthassuk.

A tömörítő algoritmus kidolgozása mellett kidolgoztuk annak hatékony gyakorlati implementálási elvét is. Megoldásunk az eredeti zlib tömörítőt a JFFS2 fájlrendszerben kicseréli egy tömörítési keretrendszerrel, amelynek működési módja konfigurálható, és beállítható úgy, hogy a blokkok tömörítésekor hívja meg az összes elérhető tömörítőalgoritmust, és azzal tároltatja el, amely a legkisebb eredményt szolgáltatta. Minden tömörített blokk mellé eltárolja azt is, hogy melyik algoritmussal lett betömörítve, így a kitömörítéskor tudni fogja melyikkel kell dolgoznia.

Ezzel a kombinált megoldással elért eredmények segítségével egy átlagos fájlrendszer image esetében 12.6-19.3%-kal jobb arányt tudtunk elérni az eredeti, csak zlib-et használó megoldáshoz képest. A módszer használatának legnagyobb hátulütője a sebesség csökkenés. A gyakorlatban ennek mértékét leginkább a boot (bekapcsolási) idő növekedése tükrözi, amely 27 másodpercről 61-82 másodpercre növekszik.

Eredményeinket a [15]-ben publikáltuk. A megvalósított keretrendszer a JFFS2 fájlrendszer és a Linux Kernel hivatalos része lett. A tömörítési algoritmus a 6,917,315-os számú USA szabadalommal [40] lett levédve.

3. Továbbfejlesztett B+ fa alapú flash fájlrendszer teljesítményjavítás

A beágyazott rendszerek túlnyomó többsége flash chipet használ háttértárként, alacsony fogyasztása és robosztussága miatt (nincs benne mozgó alkatrész). Azokon az alkalmazási területeken, ahol az adatok hatékony tárolása is szükséges módosítási lehetőséggel, ott az adatok tárolására fájlrendszert használnak.

A legtöbb fájlrendszert azonban hagyományos adathordozókra, azaz merevlemezekre tervezték, viszont a flash chipek néhány tulajdonságukban jelentősen

különböznek a merevlemezektől. Ezek miatt a legtöbb hagyományos fájlrendszer közvetlenül nem használható rajtuk, hanem csak áthidaló köztes réteg (pl. FTL) segítségével. Ez viszont teljesítményromlással jár.

Erre jelentenek hatékonyabb megoldást azok a fájlrendszerek, amelyek kifejezetten flash háttértárakra készültek. Az egyik legelterjedtebb Linux alapú flash fájlrendszer a JFFS második verziója, a már említett JFFS2 volt. Ez a fájlrendszer azonban (alapgondolatából kiindulóan) kizárólag a memóriában tárolja index információit, amit felcsatoláskor a teljes flash átnézésével épít föl. Ez 512MB fölötti flash méret esetén gyakorlatilag használhatatlan, lassúsága és memóriaigénye miatt.

Mivel a gond a fájlrendszer alapjaiban volt, ezt megoldani csak új fájlrendszer tervezésével volt lehetséges. Tervezéskor felhasználtuk a LogFS fájlrendszerben lévő flashre adoptált B+ fa adatszerkezetét, a wandering tree-t, kijavítottuk annak gyengeségeit, illetve továbbfejlesztettük, egy, a gyakorlatban is valóban használható algoritmussá, amelyet TNC-nek elneveztünk el. Főbb tulajdonságai:

- Az adatszerkezet és a hozzá kapcsolódó algoritmus magában foglalja mind a flash-en, mind a memóriában lévő adatokat és azok kezelését.
- A teljesítmény-javítást fa műveletek összevonása mellett cache-eléssel is elősegíti.
- Szükséges memóriaigénye korlátozható, illetve képes alkalmazkodni az éppen szabad memóriamennyiséghez.
- Jól tűri a beágyazott rendszereknél bármikor előfordulható hirtelen áramkimaradást.

A TNC használatának segítségével a wandering tree-hez képest a teszt környezetben a flash műveletek 98.2-99.4%-a megspórolható volt.

Eredményeink a [19]-ben kerültek publikálásra. A TNC az UBIFS és a Linux kernel hivatalos részévé vált, illetve felhasználásra került a Nokia N900 okostelefonban is.

Appendix A

The DTD of SRML

```
<!ELEMENT semantic-rules ( rules-for* ) >
<!ELEMENT rules-for ( rule* ) >
<!ATTLIST rules-for root  NMTOKEN  #REQUIRED>
<!ELEMENT rule ( expr ) >
<!ATTLIST rule element  NMTOKEN  #REQUIRED
              attrib  NMTOKEN  #REQUIRED
>
<!ELEMENT expr ( binary-op | attribute | data |
                 no-data | if-element | if-expr |
                 if-all | if-any | current-attribute |
                 position | external-function ) >
<!ELEMENT binary-op ( expr, expr) >
<!ATTLIST binary-op op  (add | sub | mul | div | exp | equal |
                         not-equal | less | greater | or |
                         xor | and | nor | contains | concat |
                         begins-with | ends-with )    #REQUIRED
>
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute element NMTOKEN                   "srml:this"
                    num     NMTOKEN                   "0"
                    from    ( begin | current | end ) "current"
                    attrib  NMTOKEN                   #REQUIRED
>
<!ELEMENT if-element ( expr, expr)>
<!ATTLIST if-element from ( begin | end ) "begin">
<!ELEMENT position EMPTY>
<!ATTLIST position element  NMTOKEN                   "srml:all"
                  from     ( begin | end )           "begin"
>
```

```
<!ELEMENT if-all ( expr, expr, expr)> <!-- cond, if, else -->
<!ATTLIST if-all element    NMTOKEN                  "srml:all"
                 attrib     NMTOKEN                  "srml:all"
>
<!ELEMENT if-any ( expr, expr, expr)> <!-- cond, if, else -->
<!ATTLIST if-any element    NMTOKEN                  "srml:all"
                 attrib     NMTOKEN                  "srml:all"
>
<!ELEMENT current-attribute EMPTY>
<!ELEMENT if-expr (expr , expr , expr ) >
                                  <!-- condition , if, else -->
<!ELEMENT data (#PCDATA) >
<!ELEMENT no-data EMPTY>
<!ELEMENT extern-function (param)*>
<!ATTLIST extern-function name NMTOKEN #REQUIRED>
<!ELEMENT param (expr)>
```

Appendix B

The XSD of SRML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:element name="srml-def">
      <xs:complexType>
         <xs:sequence>
                 <xs:element ref="database" minOccurs="0" maxOccurs="1" />
            <xs:element ref="rules-for" minOccurs="1" maxOccurs="unbounded" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="database">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element ref="tables" minOccurs="1" maxOccurs="unbounded" />
               <xs:element ref="references" minOccurs="1" maxOccurs="unbounded" />
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="tables">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element name="table" minOccurs="1" maxOccurs="unbounded">
                  <xs:complexType>
```

```
                  <xs:attribute name="name" type="xs:string" />
                  <xs:attribute name="key" type="xs:string" />
               </xs:complexType>
            </xs:element>
         </xs:choice>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="references">
   <xs:complexType>
      <xs:sequence>
         <xs:choice>
            <xs:element name="reference" minOccurs="1" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:attribute name="root" type="xs:string" />
                  <xs:attribute name="root_key" type="xs:string" />
                  <xs:attribute name="child" type="xs:string" />
                  <xs:attribute name="child_key" type="xs:string" />
               </xs:complexType>
            </xs:element>
         </xs:choice>
      </xs:sequence>
   </xs:complexType>
</xs:element>



<xs:element name="rules-for">
   <xs:complexType>
      <xs:sequence>
         <xs:choice>
            <xs:element ref="rule-def" minOccurs="1" maxOccurs="unbounded" />
         </xs:choice>
      </xs:sequence>
      <xs:attribute name="root" type="xs:string" />
      <xs:attribute name="key" type="xs:string" use="optional" />
   </xs:complexType>
</xs:element>


<xs:element name="rule-def">
   <xs:complexType>
```

```
      <xs:sequence>
         <xs:element ref="rule-instance" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="mode" default="validate" use="optional">
         <xs:simpleType>
            <xs:restriction base="xs:string">
               <xs:enumeration value="validate" />
               <xs:enumeration value="correct" />
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="match" default="any" use="optional">
         <xs:simpleType>
            <xs:restriction base="xs:string">
               <xs:enumeration value="any" />
               <xs:enumeration value="all" />
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="key" type="xs:string" use="optional" />
   </xs:complexType>
</xs:element>

<xs:element name="rule-instance">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="validation-error" type="xs:string" />
         <xs:element name="expr" type="ExprType" minOccurs="1"
            maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:complexType name="ExprType">
   <xs:choice>
      <xs:element ref="binary-op" minOccurs="1" maxOccurs="1" />
      <xs:element ref="attribute" minOccurs="1" maxOccurs="1" />
      <xs:element name="data" type="xs:string" minOccurs="1"
         maxOccurs="1" />
      <xs:element name="no-data" minOccurs="1" maxOccurs="1"
         type="xs:string" />
      <xs:element ref="if-element" minOccurs="1" maxOccurs="1" />
```

```
      <xs:element ref="if-all" minOccurs="1" maxOccurs="1" />
      <xs:element ref="if-any" minOccurs="1" maxOccurs="1" />
      <xs:element ref="if-expr" minOccurs="1" maxOccurs="1" />
      <xs:element name="current-attribute" minOccurs="1"
         maxOccurs="1" type="xs:string" />
      <xs:element name="position" minOccurs="1" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="element" type="BinaryOpTypes" />
            <xs:attribute name="from" default="begin">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:enumeration value="begin" />
                     <xs:enumeration value="current" />
                     <xs:enumeration value="end" />
                  </xs:restriction>
               </xs:simpleType>
            </xs:attribute>
         </xs:complexType>
      </xs:element>
      <xs:element name="instance-value" minOccurs="1" maxOccurs="1" />
      <xs:element name="count-children" minOccurs="1" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="name" type="xs:string" />
            <xs:attribute name="key" type="xs:string" />
         </xs:complexType>
      </xs:element>
      <xs:element name="count-siblings" minOccurs="1" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="name" type="xs:string" />
            <xs:attribute name="key" type="xs:string" />
         </xs:complexType>
      </xs:element>
      <xs:element name="reg-eval" minOccurs="1" maxOccurs="1"
         type="xs:string" />
      <xs:element name="value-ref" minOccurs="1" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="path" type="xs:string" />
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:complexType>

<xs:element name="binary-op">
```

```
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr" minOccurs="2" maxOccurs="2"
            type="ExprType" />
      </xs:sequence>
      <xs:attribute name="op" type="BinaryOpTypes" use="required" />
   </xs:complexType>
</xs:element>

<xs:element name="attribute">
   <xs:complexType>
      <xs:attribute name="element" type="BinaryOpTypes" use="required" />
      <xs:attribute name="num" type="xs:integer" default="0" />
      <xs:attribute name="from" default="begin">
         <xs:simpleType>
            <xs:restriction base="xs:string">
               <xs:enumeration value="begin" />
               <xs:enumeration value="current" />
               <xs:enumeration value="end" />
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="attrib" type="xs:string" use="required" />
   </xs:complexType>
</xs:element>

<xs:element name="if-element">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr" minOccurs="2" maxOccurs="2"
            type="ExprType" />
      </xs:sequence>
      <xs:attribute name="from" default="begin">
         <xs:simpleType>
            <xs:restriction base="xs:token">
               <xs:enumeration value="begin" />
               <xs:enumeration value="end" />
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
   </xs:complexType>
```

```
</xs:element>

<xs:element name="if-all">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr" minOccurs="3" maxOccurs="3"
            type="ExprType" />
      </xs:sequence>
      <xs:attribute name="element" type="xs:string" default="srml:all" />
      <xs:attribute name="attrib" type="xs:string" default="srml:all" />
   </xs:complexType>
</xs:element>

<xs:element name="if-any">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr" minOccurs="3" maxOccurs="3"
            type="ExprType" />
      </xs:sequence>
      <xs:attribute name="element" type="xs:string" default="srml:all" />
      <xs:attribute name="attrib" type="xs:string" default="srml:all" />
   </xs:complexType>
</xs:element>


<xs:element name="if-expr">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr" minOccurs="3" maxOccurs="3"
            type="ExprType" />
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:simpleType name="BinaryOpTypes">
   <xs:restriction base="xs:string">
      <xs:enumeration value="add" />
      <xs:enumeration value="sub" />
      <xs:enumeration value="mul" />
      <xs:enumeration value="div" />
      <xs:enumeration value="exp" />
      <xs:enumeration value="equal" />
      <xs:enumeration value="not-equal" />
```

```
            <xs:enumeration value="less" />
            <xs:enumeration value="greater" />
            <xs:enumeration value="or" />
            <xs:enumeration value="xor" />
            <xs:enumeration value="and" />
            <xs:enumeration value="nor" />
            <xs:enumeration value="contains" />
            <xs:enumeration value="concat" />
            <xs:enumeration value="begins-with" />
            <xs:enumeration value="ends-with" />
            <xs:enumeration value="equal-rounded" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Bibliography

[1] UBIFS website. http://www.linux-mtd.infradead.org/doc/ubifs.html.

[2] H. Alblas. Introduction to attribute grammars. *Springer Verlag, In Proc. of SAGA (H. Alblas and B.Melichar eds.) LNCS*, 545:1–16, 1991.

[3] R. Benes, S.M. Nowick, and A. Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. In *Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on*, pages 43 –56, mar-2 apr 1998.

[4] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Selective instruction compression for memory energy reduction in embedded systems. In *Proceedings of the 1999 international symposium on Low power electronics and design*, ISLPED '99, pages 206–211, New York, NY, USA, 1999. ACM.

[5] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, September 2003.

[6] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, April 2003.

[7] T. Bray, J. Paoli, and C. Sperberg-MacQueen. Extendable markup language, http://www.w3.org/tr/rec-xml, 1998.

[8] Jr. Breternitz, M. and R. Smith. Enhanced compression techniques to simplify program decompression and execution. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 170 –176, Oct 1997.

[9] Mario Cannataro, Gianluca Carelli, Andrea Pugliese, and Domenico Sacca. Semantic lossy compression of XML data. In *Knowledge Representation Meets Databases*, 2001.

[10] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 163–172, 2001.

[11] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[12] R. Ferenc, S.E. Sim, R.C. Holt, R. Koschke, and T. Gyimothy. Towards a standard schema for c/c++. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 49 –58, 2001.

[13] Christopher W. Fraser. Automatic inference of models for statistical code compression. *SIGPLAN Not.*, 34(5):242–246, May 1999.

[14] Minos Garofalakis, Dongjoon Hyun, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for constructing decision trees with constraints. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 335–339, New York, NY, USA, 2000. ACM.

[15] T. Gergely, F. Havasi, and T. Gyimóthy. Binary code compression based on decision trees. *Proceedings of the Estonian Academy of Sciences Engineering*, 11(4):269–285, 2005.

[16] Charles F. Goldfarb and Paul Prescod. *XML Handbook with CD-ROM*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2001.

[17] T. Gyimóthy and T. Horváth. Learning semantic functions of attribute grammars. *Nordic Journal of Computing*, 4(3):287–302, Fall 1997.

[18] F. Havasi. Increasing compression performance of block based file systems. Conference of PhD Students in Computer Science, Szeged, 2004.

[19] F. Havasi. An improved B+ tree for flash file systems. In *SOFSEM 2011: Theory and Practice of Computer Science: 37th Conference on Current Trends in Theory and Practice of Computer Scienece, LNCS 6543*, pages 297–307, 2011.

[20] Ferenc Havasi. XML semantics extension. *Acta Cybernetica*, 15(2):509–528, 2002.

[21] Ha-Joo Song Hyun-Seob Lee, Sangwon Park and Dong-Ho Lee. An efficient buffer management scheme for implementing a B-Tree on NAND flash memory. *Embedded Software and Systems*, pages 181–192, 2007.

[22] IBM. Codepack: Powerpc code compression utility user's manual version 3.0., 1998.

[23] HP Inc. ipaq h3000 pocket pc reference guide. http://h200005.www2.hp.com/bc/docs/support/UCR/SupportManual/TPM_177711-001/TPM_177711-001.pdf.

[24] Intel. Understanding the flash translation layer (FTL) specification. Technical report, Intel Corporation, 1998.

[25] Robert Mertens Jörn Engel. Logfs - finally a scalable flash file system. CSCI 390 Spring 2008 Senior Seminar.

[26] M. Kálmán, F. Havasi, and T. Gyimóthy. Compacting XML documents. *Information and Software Technology (Impact Factor 1.522)*, 48(2):90 – 106, 2006.

[27] Miklós Kálmán and Ferenc Havasi. Enhanced XML validation using SRML. *International Journal of Web & Semantic Technology (IJWesT)*, 4(4):1 – 18, 2013.

[28] D. E. Knuth. *Semantics of Context-Free Languages*, volume 2, pages 127–145. Springer-Verlag, New York, 1968.

[29] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 218 –228, 2000.

[30] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 294–299, New York, NY, USA, 2000. ACM.

[31] Mark Levene and Peter Wood. XML structure compression. 2nd International Workshop on Web Dynamics, Honolulu, Hawaii, 7th May 2002.

[32] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. *SIGMOD Rec.*, 29(2):153–164, May 2000.

[33] M-Systems. Two technologies compared: NOR vs. NAND - white paper. Technical report, M-Systems Corporation, 2003.

[34] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[35] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. Lazy-update B+-Tree for flash devices. *Mobile Data Management, IEEE International Conference on Mobile Data Management*, pages 323–328, 2009.

[36] G. Psaila and S. Crespi-Reghizzi. Adding Semantics to XML. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 113–132, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[37] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[38] J. R. Quinlan. *C 4.5: Programs for machine learning.* Morgan Kaufmann, 1993.

[39] Á. Beszédes-Á. Kiss M. Tarkiainen R. Ferenc, F. Magyar. Tool for reverse engineering large object oriented software. *SPLST*, pages 16–27, June 2001.

[40] Á. Beszedes T. Gergely, F. Havasi. Model based code compression, US patent number 6,917,315, 2003.

[41] P.M. Tolani and J.R. Haritsa. XGrind: a query-friendly XML compressor. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 225–234, 2002.

[42] J. Turley. The two percent solution. Embedded Systems Design, 2002.

[43] W3C. DOM standard. http://www.w3.org/DOM/.

[44] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, June 1987.

[45] David Woodhouse. JFFS: The journalling flash file system. In *Proceedings of the Ottawa Linux Symposium*, pages 177–182, 2001.

[46] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient B-Tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3):19, 2007.