Binary XML Transfer using Direct Compilation Techniques

Christian Horn

Experience Report and Position Paper
W3C Workshop on Binary Interchange of XML Information Item Sets
Santa Clara, California, September 24-26, 2003

Abstract

This paper describes a technique for compiling XML documents into a Virtual Machine code that constructs at execution time an optimised tree representation for the original XML document. The technique reduces the amount of data to be transferred, eliminates parsing time on the client side completely, and creates an optimised tree representation.

The technique is currently used in two publishing scenarios:

- precompilation of medium sized documents (ca. 1 MByte XML) for use in multiple clients (one-to-many-transmission) and
- the on-the-fly processing of small size documents (ca. 100 KByte) in one-to-one transmission

The elimination in parsing time reduces the client side document loading time by a factor between 2 and 10 (depending of the platform) and the bandwidth requirements by a factor of 6, while the client side memory requirements are reduced by a factor of 3 to 4.

This paper describes the technique as it has been in commercial usage for the last two years as well a proposal for a generalized and improved technique based on the experience gathered so far.

Authors Contact Address:

Lionet Ltd., 1-2 Eden Quay, Dublin 1, Ireland http://www.lio.net ch@lionet.ie

1.0 Introduction

The Extensible Mark-up Language (XML) is a wonderfully structured language, except that the lengthiness of expression makes it often a performance burden for transmission, general I/O, and parsing overhead.

Lionet Ltd. is using XML at the core of it's software since 1997. In balancing beauty and simplicity of the overall architecture with performance, memory requirements, and general practicality, we have developed (like everyone else active in the field) some technologies that stretch beyond the remits of the current XML standards. None of these “special considerations” in isolation is a silver bullet, but together they proved very powerful and enabled the binary compilation technique.

The rest of this chapter describes these special aspects that have to be taken into account to come up with an all round solution. Chapter 2 describes the binary compilation technique as it is used for the last two years. Recognising that the summary description of “how we have done it” is only of interest to a small minority Chapter 3 gives a description, of how we would do it better in the second round. We didn’t go yet the next step, because the current solution serves as reasonably well, and it became obvious that something similar was going to come from the direction of the W3C anyway. It is with pleasure that I share the views we have developed, so that they may become part of what is going to come.

1.1 Immutable Data Structures

While application software is traditionally conceived as single threaded, one has to consider that any service oriented architecture is multi threaded. This implies that traditional approaches of object-oriented software design with get/set methods create a liability, in that the (potentially) uncontrolled use of setter methods create interference between parallel threads and henceforth requires unnecessary duplication of data items.

We found that the pattern of immutable data structures is far more appropriate in the multi threaded environment. In this design pattern, the computational processes are encapsulated in what is traditionally called factory methods. Once objects have been created and are shared with the environment, they won’t change any more. As a side effect it reduces the need for synchronisation and enables the widespread use of structure-sharing, i.e. the continued use of sub trees.

Pro:

- Reduced memory requirement through re-use of sub structures.
- Streamlined processing of XML data structures in a shared memory environment, for example implementing an event driven architecture, without unnecessary formatting and parsing processes.
- Simplified implementation of Undo/Redo architecture.

Contra:

- The most obvious implication of the use of immutable data structures is that the set/add/remove methods defined by DOM have to be disabled.
- The use of structure sharing implies additional that the *getParent*() methods are not available, when several (isomorphic) sub trees are mapped onto the same tree structure. When parent references are required, they have to be traced algorithmically.

The introduction of immutable data structures supports the shift to functional programming style and is highly productive within a closed production environment. When opening up to a wider audience, the restricted use of DOM appears currently not acceptable. However, a binary interchange technique should support the potential advantages of immutable data structure and it should be at the discretion of the client to built either a restricted or full DOM representation based on the data stream.

1.2 Minor Tags and Minor Stacks.

Since we were interested in structured text documents, we found it useful to distinguish between normal XML tags and minor tags. Minor tags are tags, that are commonly used to provide additional attributes to elements of otherwise continued text, like ... and ... in XHTML. This distinction has proven very useful, since it allows to keep the integrity of continued text, while still handling the additional information.

Minor Tags may be stacked, like in .... Accordingly there is the concept of a Minor Stack. A minor stack is a list implementation of a stack of Minor tags and their attributes.

1.3 Text Blocks

The DOM model assumes that blocks of text are considered as entities, that may be split temporarily, but would revert to the maximal block size after parsing. This technique is very useful for the handling of leaf nodes of structured data where there is often the choice of representing the data either as attribute value or as text entity.

When handling unstructured (natural) text within an XML document, it seems however more appropriate to compromise on another level of granularity. We did a series of experiments to find an optimal break down for our purposes (delivery of e-Learning material) and found that a separation in words and punctuation signs with an independent spacing information, is the most effective one, since this gives the highest level of re-use, while reducing at the same time the text parsing requirements for the rendering of text blocks.

1.4 Nesting Blocks

Nesting Blocks are the equivalent of elements in DOM. We found it however more useful, to abstract from the attribute and sub node chaining and provide abstract access methods, thus hiding the actual implementation decision. It seems that DOM is revealing far to much implementation detail, thereby creating a jungle of different application styles. We found that a dramatic reduction in interface complexity has improved software quality and allowed additionally the fine tuning of the implementation for highest performance within the limited functionality. As a result we achieved simultaneous improvements in commonly contradicting areas:[1]

- Improved overall software quality,
- Reduced binary code size (measured are Java byte code), and
- Reduced run time memory requirements.

1. A DOM-lite model may be the way forward on this side.

1.5 Normalised Formatting

Initially we were using XML editing techniques that preserved the textual layout of the original XML data. This had the advantage that the differences in XML structures after a processing step were quite easily identifiable, for example using the difference tools of the operating system or the configuration management system.

Keeping the formatting information within the parsed XML files represented however an overhead that was for processing in general not required. What we required was actually only the information whether there was some spacing in front of an element or not.

For newly synthesized XML structures there was however the problem of generating an initial layout. This lead to the definition of a canonical textual representation of XML data. Once this was in place, it took about half a year, until all XML data were actually (voluntarily) converted to the normalised text format.

The practical advantages of using a unified format were as compelling as the use of strict coding guidelines within any company:

- The data files looked the same, independent which employees or tools were used to format them.
- The unified formatting allowed the unified tracking of all changes through the configuration management system.
- Once the format was synthesized automatically it was possible to use binary XML interchange without carrying the ballast of textual layout.

2.0 The Document Compilation Technique

In general we found zip compression and parsing quite adequate for transmission of small XML data files (of 40KB to 100KB). For larger documents the loading times became however unacceptable. The parsing of a 1 MB XML file within Netscape 4.7 on Solaris took however more than 40 seconds, thus generating a serious problem. When the data file was transferred simultaneously over the Internet that was just tolerable (the end user blamed the Internet and not us ;-), but when the resource file was cached locally, it was simply inacceptable.This problem triggered serious investigations into alternative transfer formats.

Since our software was pure Java anyway the idea of compiling the XML document into a Java class was actually straight forward, except the implementation took a while to reach maturity.

Table 1 gives some measurement results taken from a sample document [2]

The compiled classes provide a *getDocument*() method that constructs at run time on the client the parse tree that would normally be constructed from within the parser.

An interesting side effect of the binary transfer mechanism is, that there is actually no need to build the complete parse tree on the client side. The construction of the parse tree could be delayed until the respective nodes are being accessed, thereby reducing the run time memory requirements dramatically. When we only require the third sub tree of an XML node, we would need only to expand the top level nodes and than the third sub tree and not even expand the other sub trees. Such a technique, known as lazy evaluation, is widely accepted within the functional programming community, could benefit the usage of XML on small devices.

While for our purposes (and being a Java company) the use of standard Java byte code was pragmatically and politically the right way to go, it has some obvious disadvantages:

- It is not very appealing to non-Java folks.
- It carries a non relevant overhead, due to Java byte code requirements.
- There are inherent barriers in the Java VM specification that require work-a-rounds:
  - The byte code size for a single method is limited to 64 KByte, this requires additional levels of structuring
  - The number of local variables in a method is limited to 256.
- Some Java implementations perform extensive loading time optimization which are completely wasted on the linear initialisation code required.
- Java doesn't provide for direct array data initialisation, it requires instead lengthy linear code

TABLE 1.

Measurements

Parameter	Value
XML document size	1,284,952 Bytes[a]
zipped document size	137, 027 Bytes
compression ratio (XML : zipped document)	*9.3 : 1*
binary document size	571,294 Bytes[b]
jared binary document size	207,306 Bytes[c]
compression ratio (XML : jared binary)	*6.2 : 1*
Run time memory requirement (DOM / JDK1.4.1)	7,953,232 Bytes
Run time memory requirement (Lionet / JDK1.4.1)	2,051,968 Bytes
memory requirement comparison (DOM : Lionet)	3.8 : 1
Parsing Time (SAX / JDK1.4.1)	1,552 ms[d]
Loading Time (Lionet / JDK1.2.2)	630 ms[e]
Loading Time (Lionet / JDK1.4.1)	801 ms[f]
run time comparison (SAX parsing : Lionet loading)	*1.9 : 1*
Document compilation time (Lionet / JDK1.4.1)	3,986 ms[g]
Compile Time memory requirement (Lionet / JDK1.4.1)	3,149,104 Bytes

a. Uses canonical formatting (with 2 spaces indentation)
b. Using standard Java class format
c. By using an optimised virtual machine design, it is likely that the compressed VM code file is smaller than the zipped document file
d. Time measurements refer to an 800MHz PIII under Win2K
e. This includes the class loading and decompression time
f. Including class loading time, the degradation is due to JVM improvements
g. The current compiler implementation is not tuned towards performance.

3.0 A Generalised Technique

While the technique described in Chapter 2 is tailored for the use in connection with Java, it's shortcomings are directly related to shortcomings of the Java VM specification and particular implementations. This section attempts to distil from the experience gathered so far a platform independent encoding scheme that would allow for improved processing speed on the client and server side as well as improved compression rates to reduce bandwidth even further. The virtual machine instructions proposed below are actually derived from the code generation scheme currently used, where we are using method calls, constructors and assignments.

3.1 The XML Builder VM (XBVM)

Instead of compiling the XML structure into code for an independently specified Virtual Machine it seems better to compile into a specifically designed VM.

The arguments in favour of a specific VM are:

- The generated code has a specific structure. There is only a single entry point into a linear program. When using the lazy tree construction technique, there would be a number of entry points into a number of linear programs.
- The code is executed only once. There is no need for any form of Just-in-time compilers.
- There is only limited interference with the general execution environment. There is no need for extended byte code verification steps.

The XBVM supports 28 instructions, which are described below in detail. While the instruction sequence can be encoded using a bit-stream model, it may be more effective to use a byte-oriented structure, since the thereby resulting regular byte-level structure may be better suited to take advantages of compression techniques on the transport layer level. [1]

3.2 The XBVM Instructions

The XBVM supports 28 instructions:

- 8 instructions controlling the allocation of different tables during the construction process, namely allocation instruction for the Name Space Table (ANST), the Attribute Name Table (AANT), the Attribute Value Table (AAVT), the Attribute Table (AAT), the Tag Table (ATT), the Minor Stack Table (AMST), the Primitive Text Table (APTT) and the Substructure Table (ASST)
- 7 instructions controlling the allocation of individual objects and the entries in the appropriate table, namely for Name Space Objects (ANS), Attribute Name Objects (AAN), Attribute Value Objects (AAV), Attribute Objects (AA), Tag Objects (AT), Minor Stack Objects (AMS) and Primitive Text Objects (APT).
- 9 stack operations that control the tree construction process itself, of which three operations form the actual core (PSB, IPB, PINBA), while the other six are frequently used special cases or combinations of instructions. The introduction of these additional six introduction, allows for a further compression of the code length by about 15%.
- 2 instructions for the recording and re-use of shared substructures (RSS, ISS).
- 2 management operations (START, STOP)

3.2.1 Allocate Name Space Table (ANST)

Allocates the initial size for the Name Space table. Different tags and attributes within a document refer to different Name Spaces. The number of different name spaces used referred within a document is commonly very small.[1]

Parameter:

- Expected number of Name Space entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compilers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines. The Name Space Table supports dynamic extensions. When the actual Name Space Table size is set in advance it improves client side behaviour.

3.2.2 Allocate Name Space (ANS)

Allocates a unique object for a Name Space and sets a reference at the entry in the Name Space Table to that Name Space Object.

Parameter:

- Index of Name Space entry required (Integer). This index is in sequential code execution always the latest and would therefore not be needed. But when the lazy construction approach is used, a complete block of name spaces may have not yet been allocated (because the corresponding sub tree wasn't yet expanded). Therefore the index is always given explicitly.
- Name Space URI (String). This is the string encoding of the URI as given in the xmlns attribute.

3.2.3 Allocate Attribute Name Table (AANT)

Allocates the initial size for the Attribute Name table.[2]

Parameter:

- Expected number of Attribute Name entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compilers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines.

3.2.4 Allocate Attribute Name (AAN)

Allocates a unique object for the Attribute Name

Parameter:

- Index of Attribute Name entry required (Integer). This index is in sequential code execution always the latest and would therefore not be needed. But when the lazy construction approach is used, a complete block of attribute names may have not

1. The typical number of different name spaces we have encountered in complex documents is about 10.
2. The typical number of different attribute names we have encountered in complex documents is about 50.

yet been allocated (because the corresponding sub tree wasn't yet expanded). Therefore the index is always given explicitly.

- Index of Name Space Reference (Integer)
- Name of the Attribute (String)

3.2.5 Allocate Attribute Value Table (AAVT)

Allocates the initial size for the Attribute Value table.[1]

Parameter:

- Expected number of Attribute Value entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compilers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines.

3.2.6 Allocate Attribute Value (AAV)

Allocates a unique object for the Attribute Value

Parameter:

- Index of Attribute Value entry required (Integer). This index is in sequential code execution always the latest and would therefore not be needed. But when the lazy construction approach is used, a complete block of attribute names may have not yet been allocated (because the corresponding sub tree wasn't yet expanded). Therefore the index is always given explicitly.
- Index of Name Space Reference (Integer)
- Value of the Attribute (String)

3.2.7 Allocate Attribute Table (AAT)

Attributes are commonly implemented as lists of Attribute Nodes, each referring to an Attribute Name and an Attribute Value. The Attribute Names and Attribute Values are handled via entries into the respective tables, hence allowing the reuse of the corresponding Strings. The Attribute Table allows the allocation of unique Name Value pairs. The Attribute list is then represented as a list of integers referring to the entries in the attribute table. [2]

An optimizing compiler would place the most frequently used name-value pairs in the smaller indexes. When the list of integers is encoded as UTF-8 String, this gives the advantage, that the 128 most frequently used name-value pairs are represented by a single byte.

Parameter:

- Expected number of Attribute Value entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compil-

1. The typical number of different attribute values we encountered in documents is about 200 (obviously depending on coding style).
2. The typical number of different attribute name-value-pairs we encountered in complex documents is about 1000.

ers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines.

3.2.8 Allocate Attribute (AA)

Allocates a unique object for the Attribute Name Value Pair

Parameter:

- Index of Attribute entry required (Integer). This index is in sequential code execution always the latest and would therefore not be needed. But when the lazy construction approach is used, a complete block of attribute names may have not yet been allocated (because the corresponding sub tree wasn't yet expanded). Therefore the index is always given explicitly.
- Index of attribute name (Integer)
- Index of attribute value (Integer)

3.2.9 Allocate Tag Table (ATT)

Tags are core structuring elements within an XML document. A tag always belongs to a name space and has a unique name within that name space. For the purpose of this document, attributes are not considered part of a tag. The tag table contains references to unique objects.

Tag objects can be shared between different documents. The tag table contains only the references to the tags referred to in the current document. The consistency between different documents loaded at the same time is guaranteed by the host VM. [1]

An optimizing compiler would place the most frequently used tags pairs in the smaller indexes. When the list of integers is encoded as UTF-8 String, this gives the advantage, that the 128 most frequently used name-value pairs are represented by a single byte. Another possible compilation strategy is the batch compilation of the XML schemata and the bulk allocation of tags.

Parameter:

- Expected number of Tag entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compilers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines.

3.2.10 Allocate Tag (AT)

Allocates a unique object for a Tag.

Parameter:

- Index of Tag entry required (Integer). This index is in sequential code execution always the latest and would therefore not be needed. But when the lazy construction approach is used, a complete block of attribute names may have not yet been

1. The typical number of different tags we encountered in complex documents is about 100. This is to be distinguished from the number of tags provided in a different XML schemata invoked.

allocated (because the corresponding sub tree wasn't yet expanded). Therefore the index is always given explicitly.

- Index of name space (Integer)
- Tag name (String)

3.2.11 Allocate Minor Stack Table (AMST)

Formatting and other interpreting attributes that apply to text blocks within a continuous text run, are handled with Minor Stacks (see Section 1.2). Minor stacks are build recursively from a Minor Tag, the attributes required and a reference to a previously allocated minor stack.[1]

Parameter:

- Expected number of Minor Stack entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compilers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines.

3.2.12 Allocate Minor Stack (AMS)

Allocates a minor stack element.

Parameter:

- Index of minor stack entry required (Integer).
- Index of tag (Integer)
- Attributes (List of Integers)
- Index of Lower stack element (Integer)

3.2.13 Allocate Primitive Text Table (APTT)

Text is separated in primitive text blocks. The exact definition of a primitive text entity is irrelevant, for the moment think about words. Words are allocated in a table in such a way, that the most frequently used words get smaller indices, where as the seldom used words get larger indexes. Text blocks are then encoded as Strings in UTF-8 format with the character codes representing the entries of the respective words. [2,3]

An optimizing compiler would place the most frequently used name-value pairs in the smaller indexes. When the list of integers is encoded as UTF-8 String, this gives the advantage, that the 128 most frequently used name-value pairs are represented by a single byte.

---

1. The typical number of Minor Stack nodes we encountered in complex documents is about 100 (ca. 10 for different formatting and the rest for different hyper links).
2. We found a definition useful, that uses maximum substrings of letters and digits or single punctuation characters. An implementation that supports hyphenation may decide for a breakdown in syllables. Primitive text blocks carry then an additional bit indicating the spacing information (space in front or not) as well as a bit indicating additional formatting information. When the latter bit is set, the primitive block refers to a Minor Stack Entry with the cumulative formatting information.
3. The typical number of different primitive text blocks in a large text document is about 5000.

Parameter:

- Expected number of Attribute Value entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compilers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines.

3.2.14 Allocate Primitive Text (APT)

Primitive text blocks are allocated in bulk. This may be done either as one large block for the whole document, or a block with all the frequently used words and the rarely used words one by one.

Parameter:

- Start entry in primitive text table (Integer)
- Number of primitive text entries allocated in this block (Integer)
- The String representation for each of the primitive blocks.

3.2.15 Allocate Shared Structure Table (ASST)

The purpose of the Shared Structure Table is the re-use of complete sub trees. While it is in general quite expensive to identify all common substructures within an XML tree, domain specific compilers may be in a position to identify common substructures very efficiently.

Parameter:

- Expected number of Shared Structure entries (Integer)

Not required. On the fly compilers that generate and publish code sequentially while still processing XML data, cannot provide this size in advance. Optimizing compilers that have knowledge of the complete XML data before generating the output can provide this information to support more effective memory management on the client machines.

3.2.16 Push Sub Blocks (PSB)

The tree building process utilises a stack of building blocks. This operation allocates an array of block references, either being a primitive text block reference (see section 1.3) or a reference to a nesting block (see section 1.4), and pushes on the top of the stack of building blocks. Insert operations fill this block incrementally and a Pop operation takes off this array of block references, constructs a new nesting block from a tag, attributes, and these block references, and inserts the resulting block one level lower.

Parameter:

- Size of the block to be allocated (Integer)

3.2.17 Insert Primitive Text Blocks (IPB)

Fills the top of the stack of building blocks with a number of primitive text blocks.

Parameter:

- Number of primitive text blocks to be added (Integer)

- List of integers representing entries into the primitive block table (List of Integers)

3.2.18 Pop and Insert Nesting Block with Attributes (PINBA)

This operation constructs a new nesting block from the tag and attribute information given as parameters and the (array of) block references in the top of the stack of building blocks, removes the top of the stack of building blocks, and inserts the newly constructed nesting block at the end of the new top of the stack of building blocks.

Parameter:

- Reference to the Tag (Integer)
- Number of Attributes of this nodes (Integer)
- List of integers representing entries into the Attribute Table (List of Integers)

3.2.19 Pop and Insert Nesting Block (PINB)

This operation is a simplified version of the PINBA instruction when there are no attributes to be encoded. It constructs a new nesting block from the tag information given as parameters and the (array of) block references in the top of the stack of building blocks, removes the top of the stack of building blocks, and inserts the newly constructed nesting block at the end of the new top of the stack of building blocks.

Parameter:

- Reference to the Tag (Integer)

3.2.20 Insert Nesting Block with Attributes (INBA)

Insert nesting block operations are used to create nodes that have no further sub-blocks. In effect, an INBA operation is equivalent to a sequence of PSB 0, PINBA. It constructs a new nesting block from the tag and attribute information given as parameters and inserts the newly constructed nesting block at the end of the top of the stack of building blocks.

Parameter:

- Reference to the Tag (Integer)
- Number of Attributes of this nodes (Integer)
- List of integers representing entries into the Attribute Table (List of Integers)

3.2.21 Insert Nesting Block (INB)

The INB operation is a special case of the INBA operation when there are no attributes to be handled. In effect, an INB operation is equivalent to a sequence of PSB 0, PINB. It constructs a new nesting block from the tag information given as parameters and inserts the newly constructed nesting block at the end of the top of the stack of building blocks.

Parameter:

- Reference to the Tag (Integer)
- Number of Attributes of this nodes (Integer)
- List of integers representing entries into the Attribute Table (List of Integers)

3.2.22 Insert Primitive Text Block with Attributes (IPTBA)

The Insert primitive text block operations are the most frequently used operations as they relate to the lowest level blocks that contain only text. They are in fact equivalent to a sequence of PSB n, IPB n, PINBA. The IPTBA operation creates a new nesting block with the given tag and attributes and the sequence of primitive text objects as sub nodes and inserts the newly constructed nesting block at the end of the top of the stack of building blocks.

Parameter:

- Reference to the Tag (Integer)
- Number of Attributes of this nodes (Integer)
- List of integers representing entries into the Attribute Table (List of Integers)
- Number of primitive text blocks to be added (Integer)
- List of integers representing entries into the primitive block table (List of Integers)

3.2.23 Insert Primitive Text Block (IPTB)

The IPBT operation is a short form of the IPBTA operation for the case without attributes. They are in fact equivalent to a sequence of PSB n, IPB n, PINB. The IPTB operation creates a new nesting block with the given tag and the sequence of primitive text objects as sub nodes and inserts the newly constructed nesting block at the end of the top of the stack of building blocks.

Parameter:

- Reference to the Tag (Integer)
- Number of primitive text blocks to be added (Integer)
- List of integers representing entries into the primitive block table (List of Integers)

3.2.24 Record Shared Structure (RSS)

Records a reference to the last inserted nesting block in the shared structure table.

Parameter:

- Index of the Shared Structure entries (Integer)

3.2.25 Insert Shared Structure (RSS)

Inserts a reference to shared structure that was previously recorded in the shared structure table. When the client doesn't support structure sharing or only partial structure sharing, it should create a clone of the shared structure to the depth that is actually required.

Parameter:

- Index of the Shared Structure entries (Integer)

3.2.26 Insert Lazy Build Instruction (ILB)

Instead of generating a single long linear initialisation sequence, an optimizing compiler may decide to break that initialisation sequence into several smaller ones by creating sub sequences for certain sub trees.[1]

The ILB instruction places in an XML tree node a reference to the entry point for the XBVM that would create that node at the first access to that node, thereby delaying the allocation of objects until they are really needed.

Parameter:

- Entry point into the XBVM code (Integer)

3.2.27 Start

The start instruction initialises the environment and connects the XBVM to the host environment and allocates an array of the length 1 at the bottom of the stack of building blocks.

3.2.28 Stop

Requires that the stack of building blocks is reduced to a single element and that this element has been filled. The operation returns the reference to the block stored in this single element.

1. The strategies for break up vary from domain to domain, we found it useful to cut at the blocks with the largest number of non trivial sub trees (for example the pages in a book) or with the largest number of similar typed objects (for example subsubchapters in a thesis).

3.3 An Example

This chapter demonstrates the code generation for a simple XML structure of 1.6 KByte [3]. The code required for the binary transfer consist of table initialisation data and the actual tree building code. We leave out speculation on how the table initialisation data might be transferred best, only describing in 3.3.2 the required table initialisation and concentrate then in 3.3.3. on the actual tree building code. Under assumption that

3.3.1 The XML Source Code

FIGURE 1. XML File Material.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<material
  xmlns="http://www.pra.org.uk/material"
  xmlns:mp="http://www.pra.org.uk/material-property"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.pra.org.uk/xsd/material.xsd"
  material-id="S051">
  <general-properties>
    <trade-name>Isopar H</trade-name>
    <additional-information>Low aromatic hydrocarbon</additional-information>
    <manufacturer company-id="{68B19288-3E97-11D7-A7B5-0090276215BE}"/>
    <supplier company-id="{68B19288-3E97-11D7-A7B5-0090276215BE}"/>
    <data-sheet-uri>http://www.imperialoil.com/pdf/isoparh.pdf</data-sheet-uri>
    <cost currency="euro">35</cost>
  </general-properties>
  <physical-properties>
    <mp:material-property property-name="non-volatile">
      <mp:value>0</mp:value>
      <mp:units xlink:type="simple" xlink:href="../units/units.xml#percentByVolume"/>
    </mp:material-property>
    <mp:material-property property-name="specific-gravity">
      <mp:value>0.76</mp:value>
      <mp:units xlink:type="simple" xlink:href="../units/units.xml#specificGravity"/>
    </mp:material-property>
    <mp:material-property property-name="relative-density-of-non-volatile">
      <mp:value>0</mp:value>
      <mp:units xlink:type="simple" xlink:href="../units/units.xml#specificGravity"/>
    </mp:material-property>
    <mp:material-property property-name="solvent">
      <mp:value>100</mp:value>
      <mp:units xlink:type="simple" xlink:href="../units/units.xml#percentByVolume"/>
    </mp:material-property>
  </physical-properties>
</material>
```

3.3.2 Initialization of Tables

For this example the XBVM requires the initialisation of the following tables:

TABLE 2. **Name Space Table**

Index	Name Space
1	http://www.pra.org.uk/material
2	http://www.pra.org.uk/material-property
3	http://www.w3.org/1999/xlink
4	http://www.w3.org/2001/XMLSchema-instance
5	http://www.pra.org.uk/xsd/material.xsd

TABLE 3. **Attribute Name Table**

Index	Attribute Name	Name Space
1	"material-id"	1
2	"company-id"	1
3	"href"	3
4	"currency"	1
5	"property-name"	2
6	"type"	3

TABLE 4. **Attribute Value Table**

Index	Attribute Name
1	"S051"
2	"{68B19288-3E97-11D7-A7B5-0090276215BE}"
3	"http://www.imperialoil.com/pdf/isoparh.pdf"
4	"euro"
5	"non-volatile"
6	"simple"
7	"../units/units.xml#percentByVolume"
8	"specific-gravity"
9	"../units/units.xml#specificGravity"
10	"relative-density-of-non-volatile"
11	"solvent"

The Attribute Table appears in this example relatively senseless, since there is no apparent re-use of Attributes as name-value pairs. In other applications we have seen however very regular structures, with a large part of name-value pairs being used repeatedly. The color code used in the Attribute Table is the same as in the XBVM code example below.

TABLE 5. **Attribute Table**

Index	Name Index	Value Index	Interpretation
1	1	1	material-id="S051"
2	2	2	company-id="{68B19288-...-0090276215BE}"
3	3	3	href="http://www.imperialoil.com/pdf/isoparh.pdf"
4	4	4	currency="euro"
5	5	5	property-name="non-volatile"
6	6	6	type="simple"
7	3	7	href="../units/units.xml#percentByVolume"
8	5	8	property-name="specific-gravity"
9	3	9	href="../units/units.xml#specificGravity"
10	5	10	property-name="relative-density-of-non-volatile"
11	5	11	property-name="solvent"

TABLE 6. **Tag Table**

Index	Tag	Name Space
1	<material>	1
2	<general-properties>	1
3	<trade-name>	1
4	<additional-information>	1
5	<manufacturer>	1
6	<supplier>	1
7	<data-sheet-uri>	1
8	<cost>	1
9	<physical-properties>	1
10	<material-property>	2
11	<value>	2
12	<units>	2

TABLE 7. **Primitive Text Table**

Index	Text
1	Isopar
2	H
3	Low
4	aromatic
5	hydrocarbon
6	35
7	0
8	0.76
9	100

3.3.3 The Tree Building Code

This subsection describes the actual code sequence used for constructing the internal XML representation. For demonstration purposes, we have assumed that a byte encoding is used. Whether byte encoding or bit stream encoding is more appropriate remains to be studied. In the same way it seems likely that a tailor made compression algorithm can package the resulting code quite effectively [1].

The color coding refers to the colors used for the initialisation tables above.

TABLE 8. **The Tree Building Code**

Offset	Code	Interpretation
0	[START]	
1	[PSB] [2]	<material material-id="S051">...
3	[PSB] [6]	<general-properties>...
5	[IPTB] [3] [2] {1, 2}	<trade-name>Isopar H </trade-name>
10	[IPTB] [4] [3] {3, 4, 5}	<additional-information> Low aromatic hydrocarbon </additional-information>
16	[INBA] [5] [1] {2}	<manufacturer company-id="{68B...BE}" />
20	[INBA] [6] [1] {2}	<supplier company-id="{68B...BE}" />
24	[INBA] [7] [1] {3}	<data-sheet-uri href="http://www.imperial..." />
28	[IPTBA] [8] [1] {4} [1] {6}	<cost currency="euro"> 35 </cost>
34	[PINB] [2]	</general-properties>

TABLE 9. **The Tree Building Code (continued)**

Offset	Code	Interpretation
36	[PSB] [4]	<physical-properties>
38	[PSB] [2]	<material-property property-name="non-volatile">
40	[IPTB] [11] [1] {7}	<value> 0 </value>
44	[RSS] [1]	*record the last structure*
46	[INBA] [12] [2] {6, 7}	<units type="simple" href="..percentByVolume" />
51	[RSS] [2]	*record the last structure*
53	[PINBA] [10] [1] {5}	</material-property>
57	[PSB] [2]	<material-property property-name="spec...gravity">...
59	[IPTB] [11] [1] {8}	<value> 0.76 </value>
63	[INBA] [12] [2] {6, 9}	<units type="simple" href="..specificGravity" />
68	[RSS] [3]	*record the last structure*
70	[PINBA] [10] [1] {8}	</material-property>
74	[PSB] [2]	<material-property property-name="rel...non-volatile" >
76	[ISS] [1]	*recall* <value> 0 </value>
78	[ISS] [3]	*recall* <units type="simple" href="..specificGravity" />
80	[PINBA] [10] [1] {10}	</material-property>
84	[PSB] [2]	<material-property property-name="solvent" >...
86	[IPTB] [11] [1] {9}	<value> 100 </value>
90	[ISS] [2]	*recall* <units type="simple" href="..percentByVolume" />
92	[PINBA] [10] [1] {11}	</material-property>
96	[PINB] [9]	</physical-properties>
98	[PINBA] [1] [1] {1}	</material>
102	[STOP]	

3.4 Interfacing the XBVM with the Host VM

The XML Builder VM has however to be executed within a host VM and requires access to certain aspects of the host VM:

It requires a special DOM implementation or should provide for a DOM compliant XML access for the host VM. For highest performance and utilization of the advantages of immutable data models and the lazy generation principle, it should provide its own DOM compliant data representation. When the target environment requires the use of a predefined DOM implementation, the XBVM could directly drive a SAX builder.

When it provides its own DOM implementation, it requires the allocation and/or reference to unique objects within the host VM. The XBVM code however guarantees, that it executes only one request per unique object. Multiple uses of the same object are covered within the XBVM.

We found it useful to generate code for the host VM, that declares symbolic names for the reference objects, so that in the code itself, the XML processing can compare tags and attribute names with a simple pointer comparison.

4.0 References

1. Evans, W.S.; Fraser, C.W.: Grammar-Based Compression of Interpreted Code. *Comm.ACM* **46**(2003)8, 61-68
2. Sun Microsystem: *Developing International Java Applications.* Online Course. WZI-3100, 2003
3. KnowCoat Formulator. www.knowcoat.net