The Performance Penalty of XML for Program Intermediate Representations

Paul Anderson
GrammaTech, Inc.
317 N. Aurora St.
Ithaca, NY 14850
paul@grammatech.com

Abstract

XML has been proposed as a file format for representing program intermediate forms such as abstract syntax trees (ASTs), and dependence graphs. Although XML offers many advantages over custom binary representations of the same information, there is an associated performance cost. This paper reports on the results of experiments aimed at quantifying this cost. Two representations are studied: the abstract syntax tree, and the system dependence graph (SDG). The performance penalty for using XML is found to be very significant for ASTs, and crippling for the SDG. The tradeoff between the performance and flexibility is discussed.

1. Introduction

XML is a data format designed for describing tree-structured documents. Recently there have been several publications describing the use of XML as a data format for persistently storing program representations such as abstract-syntax trees (ASTs), control-flow graphs (CFGs), and dependence graphs [1, 3, 9, 10, 11]. Although XML offers great advantages including standards compliance and interoperability, its use as a persistent representation for forms such as abstract syntax trees and semantic graphs has a cost: they are notoriously space and time inefficient compared to custom binary representations.

This paper describes the results of a set of experiments aimed at quantifying the expense of XML-based representations relative to custom binary representations. For an XML-based representation to be worthwhile, the benefits of XML must outweigh the costs. We discuss this tradeoff and conclude that any such advantages depend strongly on the details of the application.

The remainder of this paper is structured as follows. Section 2 compares and contrasts XML formats with two different custom binary formats. Section 3 describes the design of an experiment aimed at measuring the relative performance of the formats. Section 4 gives the results of the experiment on a range of programs. Section 5 discusses whether the benefits of using XML in this domain outweigh the performance disadvantages. Finally, Section 6 describes some related work, and Section 7 presents a summary of conclusions.

2. File Formats

This section describes the details of the file formats. Section 2.1 describes options for encoding program representations in XML. Section 2.2 describes alternative custom binary representations. Section 2.3 compares these.

2.1. XML formats

Primitives Any program representation will have to represent primitive values such as strings, numbers, and booleans. In XML, strings and numbers can be easily represented by their image. However, XML does not have a built-in boolean type, so the following are all possible ways of representing `true`:

```
<booleanTrue/>
<boolean>True</boolean>
<True/>
1
```

For the purposes of brevity in the representation the encoding of `1` for `true` was used. In all such cases the context allows this to be distinguished from the integer value 1. There was no need to have an explicit representation for `false` as this is the default value of boolean-valued terms.

ASTs Abstract syntax trees lend themselves very naturally to being described in an XML format. There can be a one-to-one mapping between nodes in the AST and XML elements. For example, a simple expression `x+1` could be represented by the XML term:

```
<expr>
  <add>
    <variable>x</variable>
    <constant>1</constant>
  </add>
</expr>
```

In any serious source-code analysis tool however, ASTs are never so simple. For example, names are usually stored in a symbol table associated with each scoped region, and name references are then encoded as references to the entry in the symbol table.

Furthermore, it is normal for attributes to be associated with nodes in the abstract syntax tree. An attribute might encode the storage class of a variable, or the type of an expression. Some of these attributes will refer to other nodes in the tree.

It should be noted that the XML mechanism for specifying attributes associated with tree nodes is *not* suitable for specifying attributes of the ASTs. This is because XML does not permit attribute values to be anything other than the primitive string type. In an AST the attributes are themselves often tree-structured terms.

For these reasons it is important to associate unique identifiers with AST nodes. If an AST node is referenced by another, then that unique identifier is given as an XML attribute.

The representation of ASTs chosen for this experiment reflects this. The simple expression `x+1` is represented as follows. First the scope in which the variable is declared contains an entry for the variable `x`. A slightly abbreviated version of this is the following:

```
<cc:variable id="14">
  <source-correspondence>x
    </source-correspondence>
  <type><node id="11"/></type>
  <initializer><node id="7"/></initializer>
  <storage-class>unspecified</storage-class>
  <alignment>0</alignment>
</cc:variable>
```

The XML representation for the expression itself consists of a node with the following structure:

```
<cc:iadd id="218">
  <operands>
    <cc:expr-variable id="219">
      <variable><node id="14"/></variable>
      <type><node id="11"/></type>
    </cc:expr-variable>
    <cc:expr-constant id="220">
      <constant>1</constant>
      <type><node id="11"/></type>
    </cc:expr-constant>
  </operands>
  <type><node id="11"/></type>
</cc:iadd>
```

This defines a node of kind `cc:iadd` whose unique identifier is `218`. The one child encodes the list of operands. A `type` attribute gives the type of the expression.

SDGs The second program representation chosen for the experiment was the system dependence graph. In this representation nodes represent program points such as expressions, call-sites, parameters, and the like. There are typically two kinds of edge in a dependence graph: control dependence and data dependence.

The CodeSurfer platform [6] was used to create the dependence graph for this experiment. Before describing how to represent this in XML, it is useful to discuss the information content of the graph. CodeSurfer offers a wide range of options for the generation of dependence graphs for a program. A user can choose to omit certain phases such as global-variable analysis or summary-edge calculation, or can tune the pointer analysis along many axes. The settings used for this experiment can be summarized as follows:

- Global-variable analysis was turned on, and the information generated was retained. This means that at each point in the program, the set of all variables used, killed, or conditionally killed is available.
- Control-flow edges are retained in the final dependence graph
- Both data-dependence and control-dependence edges are computed.
- Both intraprocedural and interprocedural edges are generated.
- Summary edges are computed [8].
- The Andersen [2] pointer analysis algorithm was used, with some custom extensions to exclude nonsensical points-to sets. This is a context- and flow-insensitive algorithm. A single abstract location was used to represent all string constants.
- SSA form and π-nodes were *not* computed.
- The representation of the original abstract-syntax tree was not retained. This choice was made so that the results of this experiment would be independent of the results of the experiment comparing the AST representations.

These settings yield a very large dependence graph for some programs. For this experiment a decision was made to err on the side of completeness, as the goal was to compare representations of the same information content.

The actual settings that an end user might choose depends on the application. An application that does not need the full dependence edges will result in a much smaller dependence graph.

To represent any directed graph in XML is simple. Each node is assigned a unique id. Edges in the graph are represented as an attribute associated with a term that encodes the target of the edge. The concrete XML representation chosen for the experiment used a variant of this technique. In this representation, the SDG is a collection of Program Dependence Graphs (PDGs), one for each subprogram. Each PDG is assigned a unique integer identifier, and each node in the PDG is assigned a unique integer within its PDG. In the case where the edge is intraprocedural, the `pdg` attribute is omitted.

As mentioned above, the SDG representation includes information about abstract memory locations in the program (ABS_LOCs). Each variable gives rise to a location, as does heap memory allocated through allocators such as `malloc`. Individual fields of variables of structured types also give rise to such locations. Each program point may have up to four sets of these variables: *used* indicates those that may have their value taken at the point; *killed* indicates those whose value is changed at the point; *cond-killed* indicates those whose value is conditionally killed (e.g., through an indirect reference) at the point; and *decl-used* indicates those whose names occur at that point.

In the XML representation chosen, each ABS_LOC is also given a unique integer identifier. A section of the XML representation lists all of these, and references to them at program points is done using the identifier.

The following is an abbreviated representation of a node that represents an expression.

```
<pdg-vertex id="3" kind="expression">
  <uses><abs-loc id="2"></uses>
  <killed><abs-loc id="2"></killed>
  <cfg-targets>
    <edge target="7"/>
  </cfg-targets>
  <inter-targets>
    <edge target="77" pdg="22" kind="data">
  </inter-targets>
</pdg-vertex>
```

2.2. Custom Binary Formats

One alternative to using an XML format is to use a custom format. Binary formats are particularly attractive from a performance point of view because they can be read in from disk more quickly than textual formats. Two techniques are common:

1. When the representation is first constructed, it is written into a table in memory. This is then written to disk in large chunks using the `write` system call. Recovering the representation involves positioning a pointer into the file using `seek`, and invoking the `read` system call. If the original representation involved pointers, then these are typically adjusted after the chunk has been recovered.

2. The representation is constructed using a custom heap memory allocator that maps blocks of memory directly to a file. Recovery of the representation is as simple as mapping the file back into memory. The operating system takes care of mapping specific blocks into memory whenever they are accessed. If the representation maintains pointers, then the file must be mapped back into memory at the same address it was originally created with.

This section describes the two custom file formats used in this study. These are both used in commercial products, and have both been highly engineered for speed and efficiency. As such they are likely to represent close to the best performance that could be hoped for a custom binary format.

CIL The CIL format is a format for storing ASTs for C and C++ programs. It is the native format used by the C/C++ front end published by the Edison Design Group (EDG) [5]. This front end is widely used by many commercial compilers and software engineering tools, including CodeSurfer.

Each compilation gives rise to a single CIL file. The file is arranged as a set of scopes. An outer scope is used to represent the scope of the entire file. Each procedure has its own scope contained entirely within the file scope. The library for recovering the AST represented by the CIL file allows a user to recover the scope for each procedure independently.

The CIL format allows individual nodes to be addressed externally. A node address consists of a triple: the first item is the name of the procedure in whose scope the item resides. The second item is a byte offset into the file. The third item encodes information about the kind of the node, and allows the reader to know how to interpret the bytes found at that offset.

To recover the AST for a file, the reader uses `read` to input large chunks of the file at once, then traverses the representation translating external pointers to native pointers as necessary.

SDG The SDG format is a format for storing a whole-program system dependence graph to a file. It is the format used by CodeSurfer.

When the SDG is built, it is created in memory using a custom heap memory allocator. This has functionality

roughly equivalent to `malloc`, except that the memory allocated is mapped directly to a file. Recovering the SDG involves a simple mapping of the file back into memory using the `mmap` system call. This must be done to the same address with which it was originally mapped. Once the file has been mapped in, pointers to elements within the file are valid. A reference to a particular element can thus be achieved through a native pointer.

2.3. Comparison

The XML and custom binary representations can be compared according to the following aspects:

File Size XML representations usually consume much more space than binary representations with the same information content. Much of the overhead of an XML representation goes to the element tags. Binary representations do not need such tags, as the naming of elements is implicit. However as discussed below, this does not always hold.

There are two aspects of an XML representation that give it an advantage over binary representations. First, default values can be used to eliminate the need for some elements to be present. As mentioned earlier, if an element is of boolean type and it is not present in the XML representation, then it can be assumed to be `false`. This confers only modest savings however, unless the structure being represented is very uniform.

Second, XML is better suited to the efficient representation of elements of widely varying sizes than binary encodings. Binary encodings often require that a buffer of a fixed size be allocated for such elements. In the CIL representation, names are stored in a buffer of a fixed size. Storing a variable named `x` will use only two of these bytes, and the remainder are wasted. XML does not have this same wastage problem.

There are more opportunities for compacting data with binary representations. For example, it is common for boolean-valued elements to be multiplexed into a single word. Both the SDG and CIL representations use this technique.

Similarly, the SDG file format wastes some space because the memory allocator does not guarantee 100% utilization of allocated memory.

Memory Consumption The cost in memory to read a term into memory and reconstruct it would be expected to be high for XML representations because XML parsers typically must construct a generic in-memory representation of each term before it can be converted into an internal format. Binary representations do not suffer from this disadvantage.

Speed The time required to read in an XML term would be expected to be much higher than for binary representations. The tag must first be parsed, then checked for validity, then the data within the tag must be recursively parsed as well. Once the parsing is done, a conversion must be done to create the internal representation.

XML is particularly ill-suited to methods for reading in subsets of the terms. An XML file is required to contain only a single term. If a program needs to access any subterm, then the entire top level term must be read in.

Interoperability The greatest advantage of XML is its interoperability with other tools. Custom binary representations on the other hand are weak in this area. In order to achieve a similar interoperability, the entire file format must be published, or an API for accessing it must be made available.

3. Experimental Design

The goal of the experiment was to quantify the difference in performance between XML formats and the two custom binary formats CIL and SDG.

CodeSurfer was used as the infrastructure for these experiments. For each of the binary formats, a script was written to write out the information in the file in XML format. Care was taken to ensure that the resulting XML file had the same information content as the corresponding binary file.

To measure the performance of the XML format, an off-the-shelf DOM parser from the Apache Xerces project was used [15]. DOM parsers read in the entire XML term and create an in-memory representation of that term [17]. This was instrumented so that statistics about its performance could be collected.

To measure the performance of the binary formats two new programs were written, one for CIL and one for SDG. Each of these was designed to read in the entire file and then conduct a traversal over all elements that were read in. This traversal was considered necessary to ensure that all of the elements were fully resolved and that the data had been completely converted into the internal format. In the case of the SDG format, the traversal ensured that all pages had been mapped into memory. This allows the performance of the reader to be fairly compared to the performance of the DOM parser.

The following metrics were collected for each sample:

Nodecount the number of distinct elements in the representation.

Size the size of the file in bytes.

Used the number of bytes in use in the file. This metric is only relevant for the SDG file format, as the XML format has 100% utilization.

Utime the time spent in user code reading and reconstructing the representation.

Stime the time spent in system code during the read process.

Memory the high water mark of memory consumption after the read was completed.

The experiment was designed to be run for a number of different benchmark projects. CIL data was collected for each file in each project, and SDG data was collected for each project.

These experiments were run on a 650MHz notebook PC with 384Mb of memory running a standard distribution of Linux with version 2.4.20 of the kernel.

4. Experimental Results

The experiments were run on the benchmarks shown in Table 1. In this table, **#LOC** indicates the number of lines of code in the project,and **Files** indicates the number of source files (not counting header files).

Program	#LOC	Files	Language
tiny	3	1	C
compress	1,980	2	C
flex	13,318	13	C
hello++	8	1	C++
eLib	941	10	C++

Table 1. Benchmarks used in the experiments.

tiny is the smallest possible valid C program, consisting of a `main` with an empty body. This serves as a baseline. **hello++** is a very small C++ program that uses the `iostream` package to write `"Hello world"` to the console.

4.1. XML versus CIL

Table 2 shows the results of running the experiment comparing the performance of an XML AST reader with a CIL reader for a small set of benchmarks.

It should first be noted that although the file tiny.c contains only three lines, it gives rise to 1,821 AST nodes. The majority of these nodes come not from the program text itself, but from definitions that are predefined by the front end. For example, the front end creates ASTs for predefined types and predefined macros. This is common practice, as all modern languages have such predefined names.

Table 3 above shows the average of the collected metrics.

The first notable result is that the size of the XML representation on disk is consistently better: the size of the XML file is on average 60% of the size of the CIL file. This is a result of the fact that XML is better at representing variable-sized structures compactly than the CIL representation. In this case, the bulk of this space is taken up by strings representing program variables. For C++ programs, the CIL representation is worse. This is because of name mangling. The AST stores names of items such as classes and methods in a mangled form. As a result proportionally more of the storage goes toward representing names, and so the penalty for not having a compact representation for names is greater.

The difference in time and space to re-import the representations is very significant however. For C programs, it took on average of twenty-three times longer to read an XML AST as it did to read the same AST in CIL format. On average, importing the XML representation consumed five times as much memory.

The performance penalty for using XML to represent ASTs is significant, but it is certainly not unreasonable for all applications. Several projects have using XML for ASTs with success for many years [14]. In many cases, the time and space cost is dwarfed by much more expensive subsequent phases. In others, the analysis is very lightweight, so the cost is negligible. These results simply represent a data point that will help an engineer make a rational decision about whether to use XML.

4.2. XML versus SDG

Figure 4 shows the results of running the experiment comparing the performance of an XML dependence graph reader with the SDG reader.

It should be noted that for *tiny*, the size of the SDG files is approximately two megabytes, and that only one twentieth of this is actually used. This is because the SDG format is designed to be fast for much larger projects. A large cache is allocated when the dependence graph is first created. This is subsequently freed, but the allocator does not subsequently reduce the size of the file, so it remains at its high water mark. Larger projects can consume this freed space however, so they may have better utilization.

The amount of memory needed to read in the XML representation of the SDG is enormous.

Drawing conclusions about the scalability of the SDG representation from the SDG raw metrics for these benchmarks is not especially useful. As discussed above, the high startup cost of the SDG representation completely dominates most of these benchmarks. It is much more instructive to look at the marginal costs of the representation. That is,

			file sizes		utime		stime		memory	
File	**#LOC**	**#nodes**	**XML**	**CIL**	**XML**	**CIL**	**XML**	**CIL**	**XML**	**CIL**
Project *tiny*										
tiny.c	3	1,821	71,891	45,902	0.06	0.01	0	0	624,720	65,640
Project *compress*										
harness.c	254	9,615	383,161	1,645,286	0.25	0.01	0.02	0.02	3,221,312	1,705,232
compress95.c	1,169	13,738	473,547	391,218	0.36	0.01	0.02	0	4,191,072	394,656
Project *flex*										
misc.c	773	17,875	632,316	462,702	0.47	0	0.02	0.01	5,371,888	526,432
skel.c	1,232	26,425	889,111	725,294	0.7	0	0.04	0.02	7,955,328	787,168
main.c	988	24,451	820,740	607,982	0.63	0.02	0.04	0	7,350,608	656,240
ccl.c	149	12,683	477,445	369,206	0.33	0	0.01	0.02	3,843,744	394,016
gen.c	1,461	24,992	831,253	657,462	0.72	0.01	0.04	0.02	7,543,792	722,336
parse.c	1,706	18,866	658,569	670,562	0.51	0.01	0.01	0	5,708,528	722,512
nfa.c	709	16,958	601,372	449,970	0.46	0.01	0.03	0.01	5,103,200	460,144
tblcmp.c	888	16,884	590,354	456,570	0.43	0.02	0.03	0	5,108,928	459,856
sym.c	262	14,036	520,068	391,982	0.38	0.01	0.01	0	4,242,352	394,240
dfa.c	1,085	20,260	688,495	520,250	0.56	0.02	0.03	0	6,098,992	590,880
yylex.c	199	12,937	486,225	377,326	0.33	0.01	0.03	0	3,907,888	393,904
ecs.c	225	13,100	487,662	373,822	0.34	0	0.02	0	3,976,848	394,016
scan.c	2,723	21,931	765,851	1,530,406	0.65	0.02	0.02	0.01	6,692,768	1,575,152
Project *hello*++										
hello.cpp	8	43,179	1,595,183	3,870,998	1.21	0.04	0.08	0.03	12,882,832	3,948,960
Project *eLib*										
journal.cpp	9	28,251	1,021,501	2,133,710	0.77	0.02	0.01	0.01	8,388,240	2,171,584
user.cpp	73	49,354	1,803,982	4,344,894	1.39	0.04	0.04	0.04	14,749,472	4,308,880
common.cpp	51	28,159	1,018,915	2,135,966	0.76	0.03	0.03	0.01	8,388,736	2,170,608
loan.cpp	30	46,439	1,700,937	4,239,474	1.25	0.04	0.04	0.03	13,878,096	4,244,912
main.cpp	240	59,142	2,135,545	5,267,270	1.62	0.06	0.05	0.04	17,625,152	5,294,592
document.cpp	95	47,744	1,745,437	4,268,646	1.28	0.04	0.07	0.05	14,274,704	4,310,656
technical_report.cpp	26	45,274	1,664,828	4,096,438	1.23	0.04	0.07	0.03	13,552,592	4,113,664
book.cpp	14	27,391	994,554	1,995,762	0.7	0.01	0.06	0.02	8,189,216	2,039,328
library.cpp	189	40,982	1,473,435	3,214,402	1.11	0.03	0.06	0.03	12,463,888	3,158,864
internal_user.cpp	14	28,339	1,024,407	2,137,110	0.71	0.01	0.1	0.03	8,460,688	2,171,648

Table 2. XML vs CIL

Metric	**XML**	**CIL**	**Ratio**
File size/node	35.93	58.44	0.61
Time(μs)/node	28.26	1.22	23.08
Memory/node	302.17	59.9	5.04

Table 3. Average metrics for the XML and CIL representations of the AST.

what is the incremental cost of the representation after the initial startup cost has been paid.

This comparison in turn would have been more worthwhile on larger benchmarks, in which the effect of the large startup cost would be amortized. However, the exceedingly high cost of reading in an XML representation meant that this experiment could only be run on very small programs. As can be seen from the table, reading in the XML representation of a 13,000 line program consumed over 229 million bytes of memory. Unfortunately the same experiment on *ctags* (a project of approximately 16,000 lines) failed — the process simply ran out of memory after consuming over 300 Mb. The SDG file format however used only about 23 Mb.

Table 5 shows the size of the SDG file and the amount of memory used in it for a set of larger programs. One of the striking aspects is the non-linear cost of representing the system dependence graph. The high cost of creating a dependence graph for sendmail stems from the fact that it is a highly-complicated program with a large number of global variables and many indirect function calls.

It should be pointed out that less conservative and more precise build settings for CodeSurfer would have resulted in a smaller dependence graph. It is unlikely however that the nature of the smaller graph would be so significantly different as to make the XML format competitive with the SDG format.

Again, these results do not mean that the use of XML for all graphs is a bad idea. The nature of interprocedural dependence graphs is that they are fine-grained, highly-connected and large. It appears to be the combination of these attributes that makes them unsuited to an XML representation.

		Size			utime		stime		memory	
Benchmark	**#nodes**	**XML**	**SDG**	**SDG used**	**XML**	**SDG**	**XML**	**SDG**	**XML**	**SDG**
tiny	152	4,718	2,162,688	101,408	0.01	0	0	0	85,184	2,162,688
compress	35,085	1,100,910	2,686,976	566,064	1.37	0	0.06	0	13,955,520	2,686,976
flex	546,749	18,285,774	8,126,464	6,121,936	21.73	0	1.24	0	229,629,216	8,126,464
hello++	1,224	40,240	2,195,456	128,912	0.05	0	0	0	551,552	2,195,456
eLib	157,454	5,035,057	5,505,024	3,369,152	5.39	0	0.34	0.01	61,667,168	5,505,024

Table 4. XML vs SDG

Benchmark	**#LOC**	**SDG size**	**SDG used**
ctags	16,021	23,986,176	17,812,880
espresso	22,050	22,478,848	17,127,296
sendmail-8.7.5	40,943	197,328,896	175,655,352

Table 5. The size of the SDG for some other programs.

5. Discussion

It is clear from the results shown above that using an XML representation incurs an enormous and potentially crippling performance penalty relative to a custom binary representation.

It should be noted that these are all very small programs. If the use of XML to represent ASTs and dependence graphs is so inefficient, even for very small programs, why would a designer of an industrial-strength source code analysis system decide to use it?

5.1. Interoperability

The best answer is that XML enables interoperability with other tools. It might thus be possible for a reverse-engineering tool to be tightly integrated with a refactoring editor from another vendor. However, this noble and ambitious goal can only be easily achieved if the two tools agree to use exactly the same schema.

In practice, ASTs from different tools are quite different, and to date there is no widely-recognized standard on schemas. However, there have recently been efforts aimed at addressing this. Al-Ekram and Kontogiannis have recently proposed a language-neutral XML format for ASTs and PDGs [1].

There is also an ongoing industrial effort to standardize on a metamodel for abstract syntax trees through the Object Management Group (OMG), and even to define a "universal" AST for several languages [12]. This effort is named GASTM (Generic Abstract Syntax Tree Metamodel), and is part of the ADM (Architecture Drive Modernization) project [13]. The RFP for ASTM was formally issued in February 2005, and the deadline for proposals is May 30th 2005. Whether researchers and vendors flock to this standard remains to be seen.

For dependence graphs, there is even less agreement on schemas. Standards for graph file formats such as GXL exist and are highly useful [7]. However, these are metamodels, and there are few widely-recognized standards for schemas for particular graphs.

If different schemas are in use, then it is certainly possible to write tree-to-tree converters, and technologies such as XSLT make it relatively easy to write simple converters [19]. However, if the schemas are quite different, writing a converter can be a highly time-consuming and difficult task. Simple languages such as XSLT are attractive for small transformations, but they are not designed to be general purpose programming languages, so even simple tasks such as iteration are difficult to implement. They lack many of the modern features of programming languages, such as strong static typing, that are considered essential in any good modern language.

Even when using a DOM library in a high-level language such as C++ or Java, the typing model is very weak, thus making programming difficult and error-prone.

The best argument for XML interoperability is exemplified by the approach taken by GXL [7]. As its name implies, GXL was designed for *exchanging* graphs between tools — a tool with a GXL writer can exchange a graph with another tool with a GXL reader, regardless of any local formats that each may use. GXL is an XML sublanguage, but unlike the formats described in [1, 10], it does not prescribe schemas for specific graphs. As such, the designer of tool can choose the granularity and quantity of information that is converted to GXL. GXL works best when used to exchange relatively small chunks of data between tools for very specific purposes.

As a result, GXL has been very successful. The results

presented here should serve as a warning that a designer should avoid using GXL to exchange large, fine-grained, highly-connected graphs all at once.

5.2. Binary XML

Some of the performance disadvantages of XML may be overcome by a *binary* XML format. There are several such formats available, but there is no official standard yet. A binary standard is being developed, but it is unlikely to be finalized before 2007 [18]. Until a binary format is standardized and widely implemented, the interoperability argument does not apply.

5.3. Alternative DOM parsers

It is possible that Xerces-C++, the DOM parser chosen for this experiment displays pathological behavior when used to represent highly-connected graphs, such as the system dependence graph, and that an alternative DOM parser would have much better performance characteristics. It seems unlikely that Xerces is particularly badly implemented, as it is very widely used in industry, and many open-source projects.

It is certain that a custom DOM parser could be written that would be able to read the SDG format more efficiently, but this would severely undermine one of the reasons for using XML at all — the availability of generic tools.

6. Related Work

Several authors have proposed representing intermediate forms for language processors in XML.

The idea was first discussed in the literature by Badros in 2000 [3], who introduced JavaML for representing ASTs for Java programs. Other systems include XMLizer [11], srcML [14], CppML [9], and O2XML [16].

Maruyama and Yamamoto describe XSDML — their system for generating and analyzing an XML representation of ASTs and PDGs [10]. Some measurements of the performance of their system were reported in that paper, and they acknowledge that the time taken to generate the XML is unacceptable. They propose that existing compiler technologies be extended to generate files in XSDML, but do not address the much more serious performance penalty — the subsequent processing of the files.

Applications of these systems vary. The original JavaML system was used to produce source code documentation and simple metrics extraction. srcML has been used by Collard et al to build a system for extracting simple facts from C++ programs [4]. Wiharto and Stanski propose using O2XML for retargeting applications to different languages.

Al-Ekram and Kontogiannis argue that their framework is suitable for a wide variety of software analysis tasks [1]. A language-independent layer named *FactML* is proposed to unify XML representations for particular languages, such as JavaML and PascalML. Additional XML sub-languages are defined for representing control-flow graphs, the call graph, and the dependence graph. They propose using XML transformers to translate between these layers. Their paper presents some rudimentary statistics on the time taken to perform some key tasks, but no information is presented on how much memory the transformations require. Although they show that slicing of a single PDG can be made to work using this system, the evidence presented herein suggests that their framework would have trouble scaling to large programs with a fully-featured interprocedural dependence graph.

7. Conclusions

An experiment to compare the performance of XML representations of ASTs and dependence graphs against custom binary representations was described. The experiment was run against a number of small benchmarks. It was found that the use of XML representations carries a large performance penalty for ASTs: on average, reading the AST in XML format requires approximately five times as much space and twenty-three times as much time. In contrast, the XML file size is about 60% of the size of the custom binary representation.

The performance penalty of using XML to represent the program's system dependence graph is crippling. For the cases where it was possible to read in the XML representation using the DOM parser, the amount of memory consumed was enormous. For slightly larger benchmarks, the experiment failed because the process exhausted the memory available on the host machine. As the largest benchmark to succeed was only 13,318 lines of code, this argues that the use of XML for this kind of graph is infeasible for all but the smallest programs. It was not possible to reliably compare times for these experiments as the time taken to read the custom binary representation was consistently below the measurement threshold.

The cost of XML was weighed against its potential benefits. The best argument for using XML is still its interoperability, especially for exchanging relatively small amounts of information between tools for specific purposes.

8. Acknowledgements

The author wishes to thank Chi-Hua Chen for her help understanding the details of the CIL file format, and for outstanding technical support.

References

[1] R. Al-Ekram and K. Kontogiannis. An XML-based Framework for Language Neutral Program Representation and Generic Analysis. In *Proceedings of the Conference on Software Maintenance and Re-engineering (CSMR'05)*, pages 42–51. IEEE CS Press, 2005.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).

[3] G. J. Badros. JavaML: A Markup Language for Java Source Code. In *Proceedings of the Ninth International World Wide Web Conference*, pages 159–177, May 2000.

[4] M. Collard, H. Kagdi, and J. Maletic. An XML-Based Lightweight C++ Fact Extractor. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC03)*, pages 134–143, Portland, OR, May 2003.

[5] Edison Design Group. Compiler Front Ends for the OEM Market. http://www.edg.com/.

[6] GrammaTech, Inc. *CodeSurfer*. http://www.grammatech.com.

[7] R. C. Holt and A. Winter. A Short Introduction to the GXL Software Exchange Format. In *WCRE 2000: Working Conference on Reverse Engineering*, pages 162–171, Brisbane, Australia, Nov 6 2000.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.

[9] E. Mamas and K. Kontogiannis. Towards Portable Source Code Representations Using XML. In *Proceedings of IEEE Working Conference on Reverse Engineering (WCRE00)*, pages 172–182, Brisbane, Australia, November 2000.

[10] K. Maruyama and S. Yamamoto. A case tool platform using an xml representation of java source code. In *Source Code Analysis and Manipulation (SCAM'04)*, pages 158–167, Chicago, IL, September 2004.

[11] G. McArthur, J. Mylopoulos, and S. Ng. An Extensible Tool for Source Code Representation Using XML. In *Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 199–210, Richmond, VA, October 2002.

[12] OMG. Abstract Syntax Tree Metamodel ASTM (ASTM) final RFP. http://www.omg.org/cgi-bin/doc?admtf/2005-02-02.

[13] OMG. Architecture-Driven Modernization (ADM). http://www.omg.org/adm.

[14] Software Development Laboratory. srcml. http://www.sdml.info/projects/srcml.

[15] The Apache XML Project. Xerces C++ Parser. http://xml.apache.org/xerces-c/.

[16] M. Wiharto and P. Stanski. An Architecture for Retargeting Application Logic to Multiple Component Types in Multiple Languages. In *Fifth Australasian Workshop on Software and System Architectures*, April 2004.

[17] World Wide Web Consortium (W3C). Document Object Model (DOM). http://www.w3.org/DOM/.

[18] World Wide Web Consortium (W3C). XML Binary Characterization. http://www.w3.org/TR/2005/NOTE-xbc-characterization-20050331.

[19] World Wide Web Consortium (W3C). XSL Transformations (XSLT), Version 1.0. http://www.w3.org/TR/xslt.