Serializing C Intermediate Representations to Promote Efficiency and Portability

Jeffrey A. Meister
Department of Computer Science
University of Maryland
College Park, MD 20742, USA

May 13, 2008

Abstract

C static analysis tools need access to intermediate representations (IRs) that organize program data in a well-structured manner. However, the C parsers that create IRs are slow, and they are not available for most languages. To solve these problems, we investigate two language-independent, on-disk representations of C IRs: one using XML, and the other using an Internet standard binary encoding called XDR. We benchmark the parsing speeds of both options, finding the XML to be about five times slower than parsing C and the XDR about four times faster. We also demonstrate the portability of our XDR system by presenting a C querying tool in Ruby. Our solution and the insights we gained from building it will be useful to analysis authors and other clients of C IRs.

1 Introduction

There is significant interest in writing static analysis tools for C programs. The *front end* of a compiler or static analyzer parses a program's C source code and from this it builds a conveniently-structured intermediate representation (IR), which is the object of subsequent analysis. Typical IRs include abstract syntax trees (ASTs) and control-flow graphs (CFGs).

IRs are created in memory by the front-end, and they are discarded when the analysis completes. This leads to a few unfortunate consequences. For one, the front-end must re-parse the source code for every separate analysis program. Parsing is time-consuming, and the cost it adds to simple analyses may be unacceptable. Even worse, analyses must be written in the same language as the front-end, forcing developers to know or learn that language even if it is not well suited to their application. Since it is very difficult to create a C front-end, the result is that analysis coders are restricted to a small subset of programming languages for which high-quality front-ends already exist.

In this paper, we investigate language-independent, on-disk representations of IRs for the C programming language. Our goal is to solve the problems of portability and speed presented above; we want to enable programmers to write C analyses in the many useful languages for which a capable C parser does not exist (e.g., Ruby, Perl, Python, Haskell, ...) and allow tool developers to avoid repeatedly building IRs. We built our solution in OCaml on top of CIL, a popular and robust C front-end with a well-designed IR. CIL also translates many difficult C constructs into simpler ones, which makes its IR less convoluted and easier to work with. We developed two serialization formats for this IR: one using XML [1], a well-known general-purpose markup language, and another using the eXternal Data Representation (XDR) [2], an Internet standard binary encoding.

We measured the parsing times of IRs encoded in each of our serialization formats against two baselines: CIL parsing C source code and the OCaml standard library's marshaler decoding IRs written in its custom format. The OCaml marshaler's format is heavily dependent on OCaml internals and thus is not easily

portable, so it does not meet our needs, but it gives a reasonable lower bound on deserialization time in OCaml. Our benchmark suite is composed of 21 open-source C applications totaling approximately 2.8 million lines of code. Our benchmarking results show that the XML representation is slow and bloated—slower than parsing C by nearly five times on average. We illustrate that, since the design goals of the XML format do not mesh well with our needs, it is a poor choice for this application. In contrast, our XDR encoding is fast and compact; it is about four times faster than parsing C and close to the speed of OCaml's native format.

To demonstrate the portability of our solution, we wrote a Ruby decoder for our XDR data format. We implemented a simple querying system on top of this decoder and found that it was very easy and straightforward to do so. Along with the benchmarks, this gives us confidence that our system is both fast and portable, and so it should prove useful to authors of C static analyses and other clients of C intermediate representations.

2 Serializing the CIL Intermediate Representation

In order to serialize a C IR to disk, we first have to obtain such an IR from an existing C parser. We chose to use CIL, a front-end written in the mostly functional programming language OCaml. Our main reason for choosing CIL over other front-ends is that CIL simplifies C programs by transforming many difficult constructs into cleaner (but equivalent) ones. As a result, its IR is much simpler than that of other front-ends, since most of the ugly corners of C have been translated away.

The CIL IR is defined much like any other data structure in OCaml. For example, here is part of the type of control-flow statements:

```
type stmtkind =
  | Return of exp option * location
  | Goto of stmt ref * location
  | If of exp * block * block * location
  | Loop of block * location * stmt option * stmt option
  | ...
```

This definition expresses that a value of type `stmtkind` is either a return value, which is made up of a pair containing an optional `exp` and a `location`, or it is a goto, which is made up of a pair containing a reference to a `stmt` and a `location`, and so on.

As you can see, the CIL IR is mostly similar to an AST in its structure. However, it is not a true tree; there are cross-references (to support CIL's CFG features, which require a statement to know all its predecessors and successors) and back-references.

The CIL IR is constructed using the standard building blocks of user-defined types in OCaml. Thus, the principal challenge in serializing this data is deciding how these basic elements are to be represented. Of course, the standard primitive types are used: integers, floating-point numbers, strings, booleans, and so on. Abstractions over these types include mutually recursive variants (discriminated unions), records (structs), n-tuples, and lists. A suitable serialization format must directly implement all of these without flattening the structure of the IR. And, of course, one must be able to parse it both quickly and portably.

2.1 Serialization using XML

We initially chose XML as a serialization format because it seemed to contain all the necessary components. XML is fundamentally a language for describing tree hierarchies. The CIL IR fits roughly into this mold. To evaluate XML's potential, we developed a plug-in to serialize the CIL IR to XML. A fragment of an example serialized XML IR is given in Figure 1. This IR contains part of the famous "hello, world" program; it is equivalent to the C statement `printf("hello, world\n");`.

An examination of Figure 1 reveals how the key building blocks of the CIL IR are mapped to XML data. Primitive types are represented in the obvious textual manner. For example, the integer 264 on line 8 is

Figure 1: Printing “hello, world” in XML.

```
1  <statement>
2    <instruction kind="functionCall">
3      <functionCall>
4        <name>
5          <expression kind="lvalue">
6            <lvalue>
7              <base kind="variable">
8                <variableUse name="printf" id="264" isGlobal="true"
9                    isAddressTaken="false" isUsed="true"/>
10             </base>
11           </lvalue>
12         </expression>
13       </name>
14       <argument>
15         <expression kind="constant">
16           <constant kind="string">
17             <value>hello, world&#xA;</value>
18           </constant>
19         </expression>
20       </argument>
21     </functionCall>
22     <location file="hello.c" line="4" byte="7870"/>
23   </instruction>
24 </statement>
```

simply written as an ASCII string. Records, n-tuples, and lists are mapped to sequences of XML elements with a common parent. For example, the call to the function `printf` is represented in the CIL IR as a tuple containing the function's name and its argument. This tuple is stored in XML by encoding the name (lines 4 through 13) followed immediately by the argument (lines 14 through 20) with both contained as children inside the element `<functionCall>`. Discriminated unions are given an element that names the union and has an attribute encoding the discriminant, and then the values in that arm of the union are stored as children. For example, line 15 shows an instance of the expression union type that is a constant, and encoding of the constant follows on line 16. Simple property relationships among parts of the IR translate to attributes inside an appropriate element. For example, `printf` has global scope, and this is indicated with a Boolean attribute at the end of line 8. We handle back- and cross-references by assigning an ID to a node when it first appears and using the ID to refer to that node from then on. For example, `printf` is referred to by its ID on line 8.

Due to its popularity on the Web, almost every modern programming language is capable of parsing XML data to a sensible in-memory representation. There are then many existing tools that can assist programmers in working with this data structure; for example, XQuery implementations provide flexible database-like query facilities to extract information from the XML tree.

XML also seemed likely to take less time to parse than C source code. C parsing is a complex task; for front-ends to handle real-world programs, they must support many nuances of the language as it is used in practice. XML, on the other hand, was designed with simplicity and ease of parsing in mind, and a rigorous formal specification of the language exists. By this account, using XML ought to bring us a significant speed increase.

One negative fact about the XML IR in Figure 1 is immediately obvious: it is much larger than the equivalent C source code. When working with IRs, some size increase is expected and is perhaps unavoidable. One of the reasons that IRs are much easier to analyze than concrete program syntax is that they make

explicit a lot of structural information that is implicit in the source code. For instance, the descending chain of XML elements on lines 1-8 of Figure 1 tells us (proceeding from the inside out) that `printf` is the use of a name that refers to a region of memory, which is a kind of expression, which is used to identify a function, which participates in a function call, which is a kind of instruction, which is a kind of statement. In the source code, this much detail is not given; the programmer garners it from his knowledge of the C language and the context in which `printf` appears.

However, not all of the example's verbosity can be ascribed to the nature of IRs. Some of it is due to the XML format itself. In fact, two aspects of the format's design run contrary to our needs. First, terseness of XML markup was considered unimportant by the creators of the format. [3] We, on the other hand, require an efficient representation if we hope to achieve a reasonable increase in parsing speed. Second, XML was designed to be readable by humans. This is not useful to us, since we do not intend for our serialized IRs to be directly edited (surely, source code is better for this purpose).

The negative consequences of the XML design are quite apparent from Figure 1. Every element is given a meaningful name, even though such labels are unnecessary for unique identification. Worse, this name must be repeated in its entirety to close any non-leaf element; this extravagance may improve human readability, but it is not required by parsers, for which a single closing character would suffice. The XML syntax contains many meta-characters, including angle brackets, quotation marks, equal signs, and slashes, which serve to group tokens of the XML document in a legible fashion but do not themselves encode any data. Clearly, XML is much less succinct than a format for storing tree data structures could be.

In fact, XML's bloated nature eventually led us to abandon it as our serialization format of choice. It turns out that serializing the CIL IR to XML generates files so immense that their size increase over C far outstrips the parsing benefits gained from XML's simplicity. We provide benchmarks to substantiate this claim in Section 3.

2.2 Serialization using XDR

Having become dissatisfied with XML's performance, we decided to switch to a binary format in the pursuit of efficiency. We chose XDR, a standard for the description and encoding of data introduced by Sun Microsystems in the late 1980s. The XDR standard includes two key components: a language used for defining data structures and a binary format for representing those structures. An XDR implementation for a given programming language takes a definition of a data structure written as an XDR data description and generates functions to convert between the programming language's native representations of data and the XDR binary format.

Although XDR does not enjoy the wide popularity of XML, libraries that support it are available in many if not most programming languages. XDR is the intermediate encoding used for data that is being passed between computers in Sun's remote procedure call (RPC) implementation. Sun RPC libraries are widely available, and they must be capable of encoding and decoding XDR data to implement the protocol, so the necessary code exists and most likely needs only minor if any modifications to work with our system.

The XDR encoding is a rather simple format, and it meets our representational needs just as well as, if not better than, XML. Figure 2 contains the XDR representation of the same C statement as in Section 2.1: `printf("hello, world\n");`. The first column numbers each byte for reference, the second column displays the XDR-encoded IR one 32-bit word at a time (since all XDR-encoded values are 4-byte aligned), and the third column explains what each word means.

Inspecting Figure 2 helps reveal how the CIL IR is encoded in XDR. XDR specifies generally sensible C-like encodings for its primitive types: `signed` and `unsigned int`s are 32 bits long and in big-endian byte order, `hyper` is used for 64-bit integers, `float` and `double` follow the IEEE standard, and so on. For example, the word at byte 28 encodes the integer 264. We represent lists with XDR arrays, which are encoded as an `unsigned int` member count followed by the elements of the array in succession. For example, the list of arguments to the function `printf` begins at byte 36, where the 32-bit integer 1 is encoded to indicate that there is just a single argument, whose encoding then follows until byte 67. Strings are similarly encoded with a length followed by ASCII data, but the bytes used to encode the string are zero-padded to a multiple of 4, as shown at bytes 65–67 and 83. Records and n-tuples are stored using XDR `struct`s, which are encoded

Figure 2: Printing "hello, world" in XDR.

Byte Index	Hex Word	Comment
0	`00 00 00 00`	The statement has no labels.
4	`00 00 00 00`	It is a sequence of instructions
8	`00 00 00 01`	with only one member:
12	`00 00 00 01`	a function call.
16	`00 00 00 00`	The result is discarded.
20	`00 00 00 01`	The function name is an lvalue,
24	`00 00 00 00`	with a host consisting of a variable
28	`00 00 01 08`	whose ID is 264
32	`00 00 00 00`	and no offset.
36	`00 00 00 01`	There is one argument:
40	`00 00 00 00`	a constant,
44	`00 00 00 01`	in particular, a string,
48	`00 00 00 0d`	which is 13 characters long:
52	`68 65 6c 6c`	`h e l l`
56	`6f 2c 20 77`	`o , w`
60	`6f 72 6c 64`	`o r l d`
64	`0a 00 00 00`	`\n`.
68	`00 00 00 04`	The statement occurs at line 4
72	`00 00 00 07`	of the file with the 7-character name
76	`68 65 6c 6c`	`h e l l`
80	`6f 2e 63 00`	`o . c`,
84	`00 00 1e be`	starting at byte 7870.
88	`ff ff ff ff`	No CFG information was generated,
92	`00 00 00 00`	so there are no known successors
96	`00 00 00 00`	or predecessors.

as simply each of their elements concatenated together. For example, an lvalue is made up of a host and an offset, and the lvalue beginning at byte 20 encodes the host at bytes 24–31 and the offset at bytes 32–35. Discriminated unions are directly supported as XDR `union`s, encoded as an integral type identifying the arm that is present followed by that arm. For example, the 32-bit integer 1 at byte 44 signifies that the constant encoded there is a string constant, and the string follows thereafter.

It is immediately evident that this XDR IR is shorter than the XML IR given in Figure 1, even though they encode the same information. In fact, the XML IR is 462 bytes (ignoring indentation and newlines), while the XDR is only 100 bytes. This is due to the differences between the two formats. To illustrate, compare lines 14–20 of the XML with bytes 40–67 of the XDR. The XML's readable tags clearly explain that the constant string `hello, world\n` is the argument to the function `printf`. In contrast, it is not easy to discern the meaning of the XDR-encoded information without the explanatory comments. The word at byte 40 identifies some of the following data as a constant expression, but it is not clear how `00 00 00 00` conveys that information or where the constant ends; in fact, the same word is used at many different locations in the XDR IR with an entirely different meaning each time. An XDR decoder must handle this same unadorned binary data, so it is in a similar position; unlike an XML parser, an XDR decoder cannot extract any meta-data from the encoded values. For this reason, XDR is said to be an implicitly typed encoding. The XDR decoder needs foreknowledge of what the encoding represents in order to understand it. This foreknowledge comes in the form of an XDR data description.

We have created an XDR data description for our representation of the CIL IR. In this file, type information is assigned to the encoded data by building up composite types such as structures, enumerations, and discriminated unions, starting from primitive types. The syntax and semantics are similar to C, with a few minor exceptions. For instance, here is the type definition for a statement:

```
struct stmt {
   label             stmt_labels<>;
   real_stmt         stmt_content;
   int               stmt_cfg_id;
   stmt              stmt_cfg_successors<>;
   stmt              stmt_cfg_predecessors<>;
};
```

This data description explains why the example in Figure 2 takes the form that it does. A `struct` in XDR is encoded as each of its items concatenated together, so each statement begins with an array of `label`s (the `<>` syntax signifies a variable-length array). Since an array is encoded as an `unsigned int` length followed by each member, and since there are no labels in our example, the first 4 bytes of the XDR encoding are `00 00 00 00`. Hence the explanation of this value in the comment column. The next item in the `struct` is a `real_stmt`, which is another user-defined type. Its definition is not shown, but it is a discriminated union; it uses the `int`s from 0 to 10 to identify which kind of statement is present. Possibilities include `return` statements, `goto`s, `break`s, and so on. The `int` 0 refers to the kind of statement that is just an array of instructions, which is what we have in our example. Thus, the next 4 bytes are `00 00 00 00`, followed by `00 00 00 01` to indicate an array length of 1, and then encoding of an instruction begins. The instruction ends at byte 87; afterward, the last three items of the `stmt` appear, and the encoding ends.

A code-generating XDR implementation is available for OCaml in the Ocamlnet package's implementation of Sun RPC. [4] However, we found that this generator was not mature and robust enough to handle our large and complex data description, though we did make use of its mapping between OCaml primitive types and XDR primitive types. Thus, as we did with XML, we constructed an XDR-encoding CIL plug-in by hand. This process was not difficult. Since the encoding functions reflect the structures of the types that they encode, and those types are precisely specified by our data description and the XDR standard, this manual generation was rather mechanical. If someone wanted to repeat it to allow another programming language to work with our system, they could do so easily. The effort of doing so certainly pales in comparison to writing a C parser or even a much simpler XML parser.

One perhaps negative characteristic of XDR is that, since the smallest encodable unit is 4 bytes long and all values must be padded to a length that is a multiple of 4, some bytes of the encoded file do not store any data. There are a lot of discriminated unions in the XDR IR, and each time a discriminant is encoded, 4 bytes must be used even though 1 would suffice. This results in a waste of 24 bytes in the example encoding of Figure 2. However, the 4-byte unit size of XDR was an explicit design choice. The creators of the format wanted to avoid causing alignment problems on as many machines as possible without sacrificing too much space, so they chose 4 as a compromise. [5]

Compared to XML, the XDR format is clearly more compact. There are no meta-characters, since XDR is not a text format, and was never intended to be human-readable. No complicated parsing techniques are necessary; since the length of every item is always known before that item is encountered, the decoding process is merely a forward procession through a byte stream. Indeed, since the XDR files are small and simple, the decoding process is much faster than parsing C source code, fulfilling our speed goal. We show this with benchmarks in Section 3. Moreover, since XDR is a portable format, our system enables C analyses to be written in a wide variety of programming languages. We have built a simple C querying application in Ruby, which we describe in Section 4, to demonstrate this portability.

3 Experimental Results

In the previous sections, we have made claims about the sizes and parsing speeds of various C IR serialization formats. In short, we have asserted that XML IRs are too large and are thus too slow to parse (slower even than C source code), while XDR IRs are compact and fast enough for our needs. This section substantiates those claims with experimental data.

We have benchmarked the parsing speed of four formats. All benchmarks are done in OCaml using native

code. First, we measure the time CIL takes to parse C source code as a baseline. The other formats must improve upon this baseline significantly to fulfill our speed goal. Second, we measure the parsing time of XML using the PXP [6] parser. To keep speed and size numbers reasonable, we use a more compact XML form than Figure 1: all element and attribute names are restricted to four characters. Third, we measure the decoding time of XDR using our custom code. Fourth, we measure the time OCaml's standard library Marshal module takes to decode a serialized CIL IR in its native format. Though this format is not suitable for our use for portability reasons as we described earlier, it gives a bound on how fast one can reasonably expect to serialize and deserialize data in OCaml.

Our suite of benchmarking programs includes a range of different project sizes and coding styles. We use 21 open-source C applications that total approximately 2.8 million lines of code. For each of the 20 programs, we generate files that store the same program data in each of the four formats, and then we parse them back in, measuring the time taken to do so. We run each benchmark 11 times and take the median value.

Table 1: Parsing times of each data representation format (in seconds)

Program	C with CIL	XML with PXP	XML with Xmlm	XDR	OCaml Native
apache_1.3.1	1.795	8.394	3.865	0.444	0.152
bc-1.06	0.521	2.381	0.989	0.143	0.051
bind-9.3.4	11.103	52.552	24.030	2.434	0.731
bison-2.3	1.578	7.004	3.005	0.395	0.148
gawk-3.1.5	1.898	10.432	5.206	0.447	0.107
gettext-0.16	7.661	34.035	17.472	2.325	0.892
gnuchess-5.07	0.891	4.337	1.915	0.246	0.084
gzip-1.24	0.286	1.244	0.563	0.080	0.035
less-382	0.903	4.157	1.643	0.236	0.083
make-3.81	0.835	4.076	2.086	0.195	0.071
nano-2.0.3	0.652	3.056	1.461	0.153	0.050
openssh-4.2p1	3.398	13.640	5.753	0.847	0.367
retawq-0.2.6c	1.041	5.610	2.758	0.235	0.044
sed-4.1	0.597	3.287	1.626	0.147	0.043
spell-1.0	0.062	0.219	0.089	0.023	0.012
time-1.7	0.069	0.207	0.081	0.027	0.015
vsftpd-2.0.3	0.644	2.465	1.187	0.165	0.078
wget-1.9	1.157	6.122	2.727	0.296	0.092
which-2.16	0.090	0.338	0.152	0.031	0.016
xinetd-2.3.14	1.097	4.243	1.838	0.309	0.138
zebra-0.95	4.884	23.028	10.275	1.068	0.341

Table 1 gives the parsing times for each benchmark application in each format. Since this table is difficult to read, we also present the results graphically in Figure 3 using a stacked column chart. The ratio of the time taken to parse each serialized IR to the time taken to parse the equivalent C source code is represented by the height of each colored column. We present these ratios rather than the raw time measurements because parsing times vary wildly across our benchmark suite. Since the parsing times of the serialized IRs are highly correlated with the parsing time of the C code from which they were generated, dividing each by the time taken to parse the equivalent C serves to normalize the values and make the chart easier to read. We use the geometric mean to compute averages because the values are ratios without units. [7]

A rough look at the chart quickly justifies the general claims we have made throughout this paper. The XML representation of the data fails to improve on the parsing time of C source code. Instead, it is much slower, indeed unacceptably so: the average parsing time for an XML IR across this benchmark suite is 4.804 times that of the equivalent C source code. XDR, on the other hand, is significantly faster to parse than the equivalent source, fulfilling the speed goal we set out. The average parsing time for an XDR IR here is just 0.249 times that of C. Since all our benchmarking is done in OCaml, data stored in its native serialization

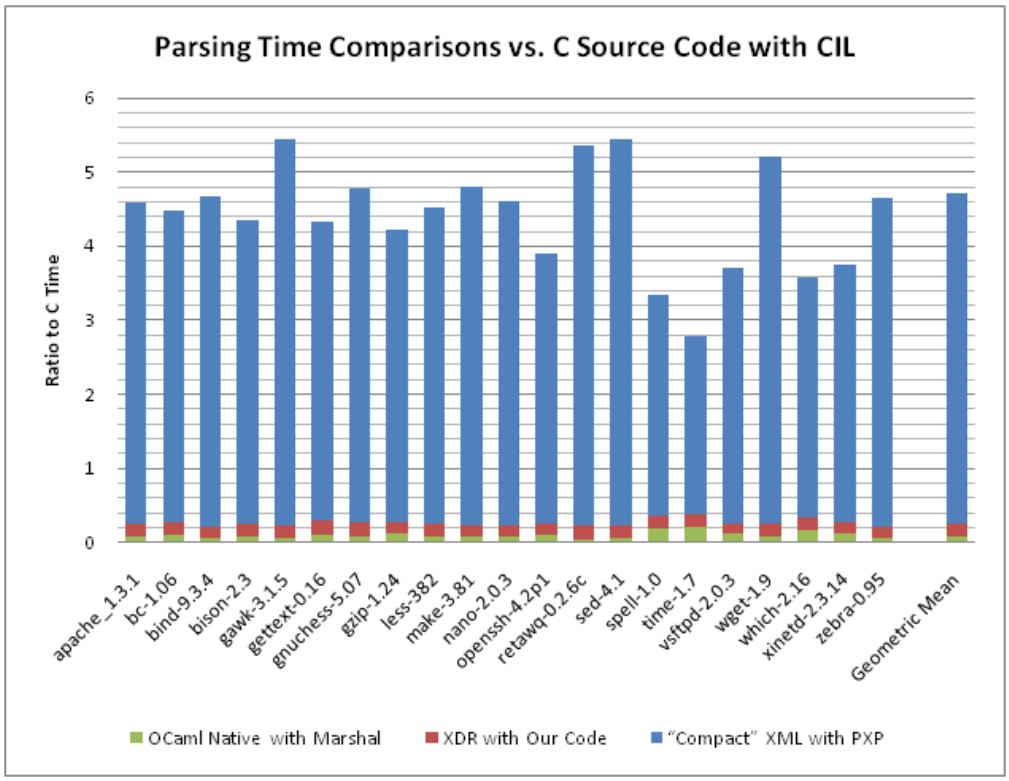

Figure 3: Parsing times of each data representation format compared to the CIL parsing time

format is understandably very fast to parse, taking an average of just 0.086 times C.

As we mentioned earlier, we were surprised to find that our XML IRs were slower to parse than the equivalent C source code. Since XML is a much simpler format than C, XML parsers are much less complicated than C parsers, so we expected that the XML format would be faster to parse. Looking at the sizes of the various formats on disk helps explain the reason for this discrepancy between our expectations and the data.

Table 2 and Figure 4 show, for each application and each serialization format, the total size of all the files used to store the IR of that application in that format. Again, we present the chart using ratios of the sizes of the equivalent C source code files. We can see that, while the XDR IRs are (as expected) larger than the C source code, they remain within a small constant factor. On average, the XDR is 2.197 times the size of C. OCaml's native format, since it can take advantage of the OCaml representation of values in memory to optimize for size, manages to be about the same size as C source code (with a ratio of 1.024) despite storing the more verbose structure of the IR. When stored in XML, however, the IRs are much larger than C code, sometimes ludicrously so: the XML for bind-9.3.4 takes up over 200 MB. On average, the XML is 19.218 times the size of C.

The massive size of the XML IRs compared to the other formats explains why XML did not meet our speed expectations. Based on the benchmark data, we can observe that XML is, as we originally expected, faster to parse than C source code. XML takes on average only 4.804 times the C parsing time despite being 19.218 times larger. In fact, from the average XML size of 35,713,000 bytes and the average PXP parse time of 9.087 seconds, we can say that PXP parses XML at roughly 3,930,000 bytes per second. This is faster than CIL parses C, which, from an average size of 2,488,000 bytes and average parse time of 1.960 seconds, is roughly 1,269,000 bytes per second. Unfortunately, for reasons explained in Section 2.1, XML is a very bloated format. In our case, when using XML to store IRs, the massive size of the files overwhelms any

Table 2: Total file sizes for each format (in bytes)

Program	C Source	XML	XDR	OCaml Native
apache_1.3.1	1,964,677	32,837,492	4,152,908	2,020,699
bc-1.06	451,429	8,093,914	1,225,216	564,373
bind-9.3.4	14,188,992	205,448,606	24,807,276	12,633,392
bison-2.3	1,521,722	24,605,255	3,384,740	1,592,645
gawk-3.1.5	2,223,308	45,311,860	5,174,064	2,383,933
gettext-0.16	14,289,641	144,300,567	15,855,152	7,992,899
gnuchess-5.07	761,395	16,376,041	2,109,972	1,011,732
gzip-1.24	243,460	4,653,581	553,220	264,487
less-382	760,468	13,371,473	2,063,308	913,634
make-3.81	934,334	17,890,733	1,909,944	915,161
nano-2.0.3	751,279	12,607,386	1,556,912	748,815
openssh-4.2p1	3,211,269	46,809,304	6,531,064	3,340,054
retawq-0.2.6c	1,337,225	24,378,488	2,909,692	1,298,890
sed-4.1	755,677	14,402,019	1,558,000	771,950
spell-1.0	42,567	659,812	94,196	49,264
time-1.7	35,973	561,834	87,924	38,151
vsftpd-2.0.3	698,526	10,165,433	1,109,452	585,814
wget-1.9	1,195,433	23,230,414	3,059,488	1,456,637
which-2.16	67,925	1,230,841	140,220	67,312
xinetd-2.3.14	1,011,404	14,908,391	2,038,892	1,030,406
zebra-0.95	5,798,649	88,130,549	11,397,796	5,832,306

speed benefit gained by using a simpler format than C with simpler, faster parsers.

Similar reasoning can be applied to the XDR data but with the opposite conclusion. With an average size of 4,367,592 bytes and average parse time of 0.488 seconds, our XDR decoder processes about 8,949,983 bytes per second. This rapid parsing speed, combined with the reasonably small size of the XDR files, makes IRs stored in XDR faster to parse than the equivalent C source code.

4 Portability Demonstration and Experience

To demonstrate that our XDR representation is portable, we have implemented a simple Ruby XDR parser for our C IR. We built this parser manually, since Ruby happens to be one of the few languages without XDR support. Fortunately, this task was not difficult—our demonstration program is just under 2400 lines of code in three Ruby files. This program shows that not only are the XDR IRs portable to languages that inherit all the machinery to deal with XDR from Sun RPC, but that even languages without a Sun RPC library can easily be made to work with our system. Programmers wishing to add support for our system to a new language can follow this program's basic strategy.

The first step, constructing a mapping from XDR primitive types to Ruby types, is simple in Ruby. The `unpack` method of `String` does almost all of the work for us. For example, to decode an XDR `double`, the procedure is simple: read 8 bytes of input into a `String`, invoke `unpack` on it with the parameter `"G"`, and a Ruby `Float` object is extracted. All the other primitive types are handled in the same way. Similar steps apply to any language that is capable of extracting integers, floating-point numbers, and so on from string data and converting them to the native representation of those types in that language. Arithmetic manipulations may be required in some cases (for instance, if native integers are not stored in big-endian byte order), but these are straightforward to handle.

The second step, building a data structure to hold the IR, should not be overly difficult for any skilled programmer in the language. The custom types in the XDR IR are built using just three means of organization: arrays, structures, and discriminated unions. Almost all languages have conventional analogs for these forms. In Ruby, we have mapped XDR arrays to Ruby Arrays, XDR structures to Ruby classes (members of

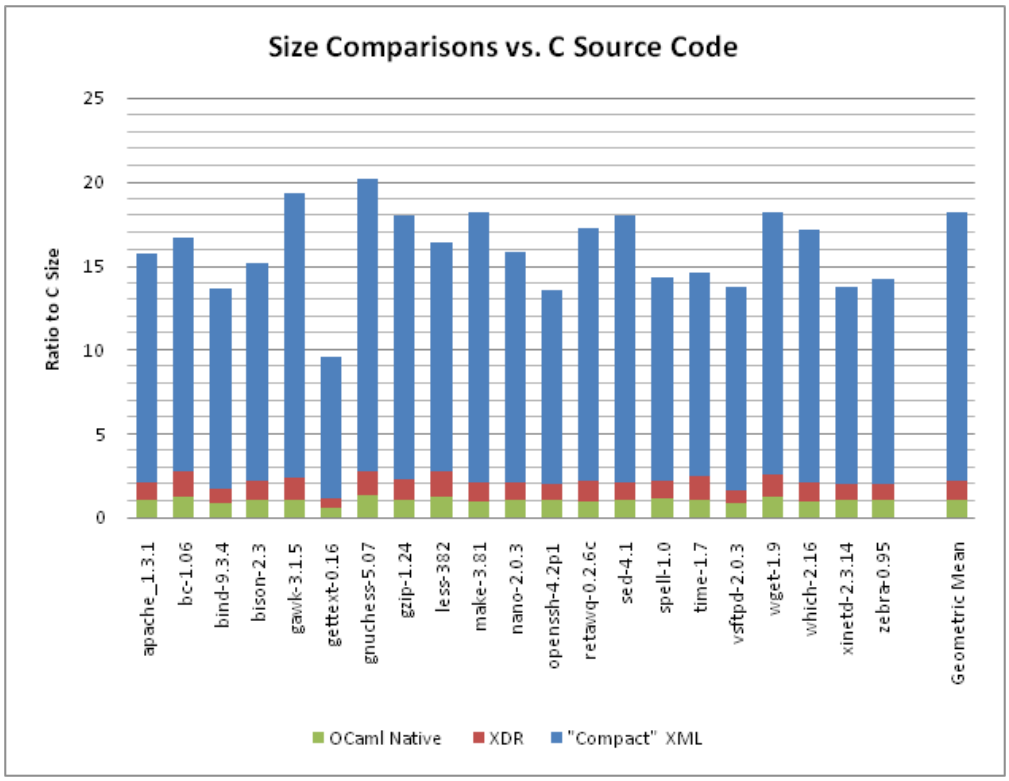

Figure 4: Total file sizes for each format compared to the equivalent C source code sizes

the structure become fields of the class), and XDR unions to Ruby inheritance from a common superclass. Once this higher-level mapping is decided upon, the conversion from XDR is mechanical; the data description file shows exactly what each custom type in the IR contains. Write functions (likely recursive ones) to walk through the XDR file, applying the mapping to each type encountered, and the decoder is complete.

Having built a decoder for XDR IRs by following this method, we proceeded to add features to show how it might actually prove useful to Ruby programmers. Since we constructed a class hierarchy, we added a variant of the visitor pattern to traverse it. Using that, we implemented visitors to perform a few basic queries on the IR: print all the function calls in a given function, print all the calls to a given function that occur anywhere, find all the global variable uses in a given function, and print all the uses of a given global variable that occur anywhere. For example, here is the complete source code for a visitor that finds all the calls to a function:

```
1  class CallsToFuncVisitor < NilVisitor
2    def initialize(func_name)
3      @func_name = func_name
4      @func_calls = []
5      @current_func = ""
6    end
7
8    def visit_global(global)
9      if global.is_a?(AST::FuncDefn)
10       @current_func = global.name
11     else
12       @current_func = ""
13     end
```

```
14    end
15
16    def visit_instr(instr)
17      if instr.is_a?(AST::FunCall) && instr.name.identifier == @func_name
18        file = instr.loc.file
19        line = instr.loc.line
20        @func_calls << "#{@current_func} @ #{file}:#{line}"
21      end
22    end
23  end
```

If an instance of this class is created by invoking `CallsToFuncVisitor.new("foo")`, it can be passed to the `accept` method at the root of our Ruby IR. Its variable `@func_calls` will then contain the function, file, and line number in which each call to `foo` appears in that program. The code is easy to understand; it simply keeps track of which function it has most recently entered, and when it encounters an instruction that is a function call to `foo`, it records that instruction's location. The visitor pattern takes care of traversing the IR and ensures that every instruction is visited and checked.

It would not be complicated for us or others to create more querying visitors. Essentially, we have built a rudimentary framework for working with C IRs that is similar to the CIL API. Of course, the CIL API is much more mature, but we have laid down the necessary foundation upon which all its features could be implemented, and we were able to do so very easily, as evidenced by the paragraphs above. Using our serialization system, we make all this capable in Ruby, where no C parser exists or is likely to exist in the foreseeable future.

5 Related Work

5.1 Persistent Storage of Parse Data

Several applications are similar to ours in that they parse program source code to discover properties about it and then store those properties in files on disk for repeated access later. Saturn [8], a software analysis tool developed by researchers at Stanford University, serializes C IRs as part of its toolchain. It stores relations garnered from parsed C code in syntax databases, and it then runs analyses over these databases, producing summary information and error reports. Saturn also uses CIL as its C frontend, and the information in the syntax databases is stored in OCaml's native marshaling format. Unfortunately, since this format is highly OCaml-specific, Saturn's syntax databases are not portable to other programming languages as ours are. Saturn analyses must be written in the Calypso domain-specific language.

The classic source code browsing tools ctags [9] and cscope [10] both parse programs and store some basic data about them in persistent files. Ctags creates an index that locates important language objects in source files, which is typically used to help text editors search for those objects. Cscope also identifies items such as variable and function names and provides querying capabilities similar to our Ruby demonstration program presented in Section 4. Ctags has parsers for 34 programming languages, and cscope can support languages with a C-like syntax, since it uses a single fuzzy parser. However, none of the parsers in ctags or cscope are anywhere near as complex as a mature C parser intended for compilation or analysis, so the data that they store on disk is not as full and accurate as the IRs serialized by our system.

5.2 Tools to Simplify Analysis Authoring

Some projects are also related to ours because their goal is to make it easier for developers to write custom analysis programs. The Program Query Language (PQL) [11], developed at Stanford University, frees developers from having to work directly with a complicated IR to write their analyses. Instead, PQL provides its own high-level language in which users describe the code patterns that they wish to locate. PQL then automatically generates static and dynamic checkers for these patterns. GEN++ [12], created at

the University of California, Davis and Bell Laboratories, similarly allows developers to build analysis tools using a custom domain-specific language designed for that purpose, and then it generates the actual analysis programs from these specifications. PQL and GEN++ represent an alternative approach to handling the problems that arise from having to work with the IRs of complex parsing frontends: automatically generate the actual analysis code to allow developers to work at a higher level.

Another vein of relevant research explores general querying capabilities on data structures. The .NET Language-Integrated Query (LINQ) [13] project at Microsoft builds in facilities for querying XML using XQuery and relational databases using SQL to the .NET family of programming languages. Because the LINQ foundation is extensible and the standard query operators work over any `IEnumerable<T>` information source, it may be possible to extend LINQ to handle C IRs. Its querying APIs would then be powerful tools for writing C analyses.

6 Conclusion

We have introduced the problems of parsing speed and analysis portability of C programs, and we have investigated on-disk language-independent representations of C IRs as a solution to these problems. We developed serialization formats for C IRs in XML and XDR. We benchmarked the parsing speeds of both formats and found that the XDR met our needs while the XML was far too slow. Finally, we built a querying tool for C programs in Ruby using our XDR format to demonstrate its portability. Our results show that our system using the XDR format solves the issues posed. We have made our code available, and we anticipate that it will be useful for developers. The insights we gained should also be helpful to anyone solving a similar problem involving efficient serialization of tree-like data.

References

[1] W3C XML page. http://www.w3.org/XML/ [29 April 2008]

[2] RFC 4506: External Data Representation Standard page. http://tools.ietf.org/html/rfc4506.html [29 April 2008]

[3] XML 1.0 Specification page, section 1.1. http://www.w3.org/TR/2006/REC-xml-20060816/ [29 April 2008]

[4] Ocamlnet page. http://projects.camlcity.org/projects/ocamlnet.html [29 April 2008]

[5] RFC 4506: External Data Representation Standard page, section 5, item 4. http://tools.ietf.org/html/rfc4506.html [29 April 2008]

[6] PXP page. http://projects.camlcity.org/projects/pxp.html [29 April 2008]

[7] Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach* (4th edn). Morgan Kaufmann: San Francisco, 2007; 34.

[8] Aiken A et al. *An Overview of the Saturn Project.* In *PASTE*, 2007.

[9] Exuberant Ctags page. http://ctags.sourceforge.net/ [29 April 2008]

[10] Cscope page. http://cscope.sourceforge.net/ [29 April 2008]

[11] Martin M, Livshits B, Lam MS. *Finding Application Errors and Security Flaws Using PQL: a Program Query Language.* In *OOPSLA*, 2005.

[12] GEN++ page. http://www.cs.ucdavis.edu/ devanbu/genp/ [29 April 2008]

[13] The LINQ Project page. http://msdn.microsoft.com/en-us/netframework/aa904594.aspx [29 April 2008]