LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-475779

Multi-language Struct Support in Babel

D. Ebner, A. Prantl, T. G. W. Epperly

March 24, 2011

Parallel Architectures and Compilation Techniques (PACT)
Galveston Island, TX, United States
October 10, 2011 through October 14, 2011

Disclaimer

Multi-language Struct Support in Babel

Dietmar Ebner
Google, Inc
ebner@google.com

Adrian Prantl and Thomas G. W. Epperly
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California
{adrian|epperly2}@llnl.gov

***Abstract*—Babel is an open-source language interoperability framework tailored to the needs of high-performance scientific computing. As an integral element of the Common Component Architecture (CCA) it is used in a wide range of research projects. In this paper we describe how we extended Babel to support interoperable tuple data types (structs). Structs are a common idiom in scientific APIs; they are an efficient way to pass tuples of nonuniform data between functions, and are supported natively by most programming languages.**

Using our extended version of Babel, developers of scientific code can now pass structs as arguments between functions implemented in any of the supported languages. In C, C++ and Fortran 2003, structs can be passed without the overhead of data marshaling or copying, providing language interoperability at minimal cost. Other supported languages are Fortran 77, Fortran 90, Java and Python.

We will show how we designed a struct implementation that is interoperable with all of the supported languages and present benchmark data compare the performance of all language bindings, highlighting the differences between languages that offer native struct support and an object-oriented interface with getter/setter methods.

I. Introduction

Babel [1] addresses widespread interoperability requirements of high-performance scientific applications that are mainly caused by *(a)* the overwhelming amount of legacy code still in use and *(b)* the trend to integrate various mathematical models, usually implemented by different teams in different languages, in order to increase simulation precision, *e. g.*, climate models might be combined with social models to predict emissions of carbon dioxide. Developing a common language ecosystem for all components is for most applications infeasible, both for technical and economical reasons.

One paradigm to manage this complexity is component based software design. This approach can greatly facilitate reuse, interoperability, and composability of software. Consequently, it has become very popular in the design of business applications and internet technology and there is large number of widely available frameworks, *e. g.*, CORBA/CCM [2], [3], Microsoft's (D)COM [4] and .Net [5], or Sun's JavaBeans [6]. The Common Component Architecture (CCA) [7] is a joint effort by researchers from both academia and U. S. national laboratories to establish and adapt these techniques for scientific computing. The CCA basically mediates how components interact with each other and with the underlying framework.

This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-XXXXXX

Babel is based on the scientific interface definition language (SIDL), which builds on previous work such as CORBA [2] or COM [8] by tailoring the idea to the needs of scientific computing. SIDL provides a language-independent object oriented programming model and type system. This allows components to share complicated data structures such as multi-dimensional arrays, interfaces, or exceptions across various languages. Babel generates the necessary glue code that maps these high-level interfaces to a particular language ecosystem. As such, it can be used stand-alone or as part of the full CCA [9] component framework, which provides additional capabilities such as dynamic composition of applications.

One of Babel's main design principles is to scale well with a growing set of supported languages. Currently, backends are available for most traditional languages relevant to the high-performance computing community including various versions of Fortran, C/C++, Java, and Python. While the main focus is on fast in-process communication, there is also full support for transparent remote method invocation (RMI) [10]. In the latter case, caller and callee may reside in a different address space or on different machines.

With its focus on high-performance computing, Babel has first-class support for fundamental numeric types and multi-dimensional arrays including array strides, dynamic ranges, and ordering specifications (row-major vs. column-major). In this paper, we discuss why and how tuple data types have been added to this mix and how they map to native language constructs such as structs, records, or derived types (Section II–IV). Section V discusses performance and code size consideration for supported language bindings.

II. Design Principles

SIDL structs are user-defined data types that map to corresponding language constructs found in most imperative programming languages, *e. g.*, structs in C/C++, records in Pascal, or derived types in Fortran. They represent a useful alternative for classes whose main purpose is to group semantically related data together; see Figure 1 for an example in SIDL. The *mode*-attribute `in` before the argument types in the example specifies that the argument is passed by value. It serves a similar purpose as the `intent(in)` attribute in Fortran. Other mode attributes are `out` and `inout`.

```
class Date {
  int getMonth();
  void setMonth(in int month);
  int getDay();
  void setDay(in int day);
  int getYear();
  void setYear(in int year);
}
```

```
struct Date {
  int month;
  int day;
  int year;
}
```

Fig. 1. Comparison of a simple SIDL struct and an equivalent class declaration.

The use of an explicit class has the main advantage that the actual storage layout is hidden in the class and can easily be changed without effects on dependent code. Structs, on the other hand, can be accessed faster in most languages and require less code to be written, thus effectively reducing development effort.

The particular design choices for the addition of SIDL structs were governed by the following goals:[1]

- **Performance** A method call to getter and setter routines for a SIDL class is always a virtual function call that, in general, involves dynamic dispatch and marshaling of arguments and return values. For regular Babel classes, this overhead is usually easily amortized over the amount of work in the procedure. However, getter and setter routines execute very small amounts of code. Thus, the overhead compared to natively supported structs can be substantial.
- **Development Effort** Regular babel classes are more verbose than structs. They (a) require the declaration of access methods for each data member, (b) have to be implemented by the user, and (c) are often less concise in their use compared to the clean syntax usually provided for field accesses.
- **Completeness** Babel tries to map SIDL constructs and types to a particular language in a way that makes experienced developers feel at home. A struct feels often more natural for a particular purpose than a fully-fledged SIDL class.
- **Compatibility** Compatibility is important in two aspects. First, related systems such as CORBA [2], [3] or WSDL [11] already support the concept of structs; SIDL structs thus facilitate the development of compatibility layers. The second aspect is compatibility of user-defined SIDL interfaces for existing legacy software. The addition of SIDL structs often allows to wrap these interfaces with little or no code. This is not only faster, it also feels more natural to people familiar with the existing interface.

SIDL structs have been designed with these considerations in mind. In particular, they support fast data exchange between C/C++ and Fortran 2003. The latter is a new Babel binding supporting Fortran 2003 features such as ISO `Bind(C)` compatibility and type extension.

SIDL structs may contain any SIDL type including arrays, and raw arrays (*r-arrays*). In particular, structs can be nested inside of other structs. *R-arrays* are a special SIDL feature that allows for low-level access to numeric arrays (*cf.* Section III).

There is currently no support for arrays of structs. While this would be possible, the implementation is non-trivial and we found that feature not heavily requested by users. Future extensions of Babel might change this however.

For regular classes, memory is automatically allocated and freed by Babel via a reference counting scheme. This is important as Babel applications often contain modules written in a variety of languages with different approaches to memory management and garbage collection. However, there is no reference counting for structs. It is the responsibility of the programmer to make sure that memory is properly allocated and released and that there are no dangling references once a struct is freed. Babel generates corresponding support functions in order to do so for languages without dynamic memory allocation such as Fortran 77.

All Babel objects support transparent remote method invocation (RMI) [12]. This means that the user's code stays exactly the same, no matter if an object is local or remote. For each struct, Babel therefore generates a serialization and de-serialization routine that assists in marshaling data for wire transfers. This code is automatically generated as a part of the client stub and does not require user modifications.

III. Babel Architecture

At its heart, Babel is a command-line tool that compiles SIDL interface definitions into glue code that is generated in one of the seven supported languages.

Babel provides a traditional object-oriented programming model with single inheritance and interfaces that can have multiple implementations. By default, all functions are virtual, *i. e.*, a function being called always depends on the dynamic type of the associated object rather than the static type of the object's reference. Babel also provides implicit reference counting and memory (de-)allocation.

Restricting Babel to the least common denominator across the whole set of supported languages would be a non-practical approach. Instead, Babel tries to take advantage of native language features such as built-in data types or method overloading whenever possible and provides reasonable alternatives in the remaining cases, *e. g.*, overloading symbols is supported in most object oriented languages while unique identifiers are required for earlier dialects of Fortran. Across all supported languages, Babel provides sophisticated features such as transparent support for remote method invocation, overloading, inheritance, and exception handling, *e. g.*, it is common use to derive a Python class from a class written in Fortran to overwrite a subset of the member functions.

In order to achieve this, Babel employs a C-based *intermediate object representation* (IOR). The IOR is exactly the same, no matter which language has been used to implement or invoke a particular method. The term "object" in IOR refers to all of the supported data structures, including structs and enumerations, rather than just objects in the sense of object-oriented programming. The SIDL language uses the term "class" to describe the latter.

[1]Note that their implementation varies widely among the set of supported Babel languages; details are discussed in Section IV.

The IOR is essential to achieve scalability across a growing set of languages. Any language binding essentially needs to translate from and to Babel's IOR thereby achieving full interoperability with all other supported languages.

Under the hood, the IOR corresponds mainly to the storage layout and calling conventions used in C. The reasons are twofold. First, C allows for fine grained control about how things are laid out in memory. Second, with few exceptions, most notable earlier Fortran standards (Fortran 95 and earlier), almost all languages support some kind of C compatibility layer effectively making C the lingua franca amongst programming languages. The IOR representation of a (class-)object is the *entry point vector* (EPV), which is a record containing function pointers to all the methods of the object. This is comparable to a virtual function table in C++.

Figure 2 depicts the control flow of a local Babel function call. On the client side, a so-called *stub* is generated that converts arguments to Babel's IOR representation, calls the proper method entry point from the object's EPV, and—if necessary—converts return values to the representation used in the original language. On the server side (*skeleton*), the inverse operations are performed, *i. e.*, arguments are converted from IOR to the particular implementation language, the user-supplied implementation is called, and return values are converted back to Babel's IOR. In addition, the skeleton is responsible to catch exceptions thrown in the implementation and convert them to a language-independent representation.

Arrays in Babel come in two different flavors:

- *SIDL arrays* are managed by the Babel runtime and are available in all variations of shape and dimension and stride. In many languages, access to the array elements is provided via a function interface; some languages bindings also provide a native interface to access the array's elements.
- *Raw arrays (r-arrays)* are a low-level alternative to the fully-fledged SIDL arrays that direct allow access to the underlying data structures. They provide the trade-off between comfort and performance. For example, in C, a one-dimensional raw single-precision array will be represented as `(float *)`.

Unlike regular SIDL arrays, *r-arrays* adhere to several constraints. Among other things, they must be contiguous blocks of memory organized in column-major order. Also, they can only be passed in `in` or `inout` mode and must retain their shape across method invocations. These restrictions also apply to structs containing *r-arrays* either directly or indirectly via another nested struct. The Babel compiler makes sure that these limitations are satisfied at compile time.

Raw arrays can be either of constant size or dynamically sized. If a dynamically sized array is passed as argument to a function, Babel requires the size of the array to be calculable from a symbolic expression statically defined in the SIDL interface definition. This size expression may only contain arithmetic operators, constant values and other (integer) arguments of that function call. If an *r-array* is a field of a SIDL struct, the requirement is that the size-expression may only refer only to (integer) fields of the same struct. This way structs are self-contained and can be passed as arguments to function calls. Because of the memory management restrictions for *r-arrays* mentioned above, structs containing *r-arrays* (either directly or via a nested struct) cannot be used as return values of functions or as `out`-arguments. The Babel compiler will automatically reject such functions. If such a behavior is desired, regular SIDL arrays should be used instead.

IV. Language Bindings

The implementation of the language bindings in Babel differs in respect of performance, convenience and level of integration with the host language ("nativeness"). Table I gives a high-level overview of the different struct implementations. More details on the implementation of arrays inside of structs are given in the comparison chart in Table II. The following paragraphs discuss all the language bindings in more detail.

A. C

In the C language, the raw IOR is presented to the user (a `struct`). In terms of performance, this is the baseline. Since the IOR coincides with the native representation no conversions are necessary and no performance penalty needs to be paid.

Figure 3 shows what happens when a C client calls a server also implemented in C. When the user writes a Babel method invocation, the client-side stub is invoked. The stub performs an indirect call of the method via the EPV. Since the stub is so tiny, Babel generates it as `inline`-attributed function, such that the only overhead is the cost of the indirect function call, which will be inserted by the C compiler in lieu of the Babel method call written by the user.

For local calls, the server-side skeleton is not needed and the EPV points directly to the server implementation. In a remote call, the EPV points to a function that serializes all arguments and pushes them over the network. On the server side, the reverse actions are performed prior to calling the user's server implementation.

B. C++

C++ is practically a superset of C that covers almost 100% of the language. For the SIDL-struct implementation this has the implication that no performance penalty is paid for conversion. To provide the programmer with a more object-oriented representation, Babel generates a C++ class[2] that inherits from the IOR. The class has a constructor/destructor pair, access functions and also defines an assignment operator such that creating copies of the C++ wrapper class can be created easily. The class also has methods used by the RMI functionality to (de-)serialize the struct from/into a string. An example C++ class for the struct in Figure 1 is shown in Figure 4. The use of the field access functions is optional—it is still possible to access the publicly inherited fields directly—but, if used, they convert the IOR data types into their C++ equivalents: For example, `(char *)`-strings are converted

[2]Technically it is also a `struct`, such that all the fields of the parent struct remain `public`.

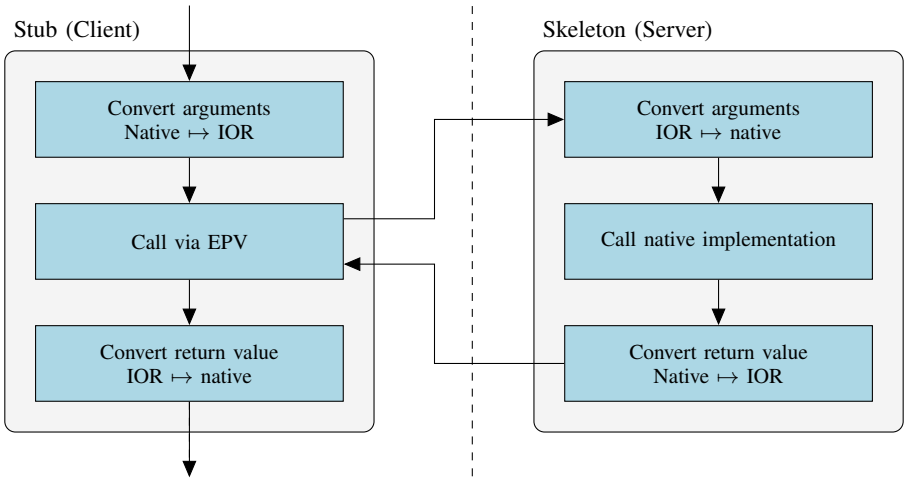

Fig. 2. Babel method invocation. Arguments and return values are converted to Babel's intermediate object representation (IOR) before being passed.

Language	Appearance	Argument passing	Field access	Implementation approach
C	native	fast	fast	`struct` (direct access to IOR)
C++	native	fast	fast	like C
Fortran 2003	native+functions	fast	fast (mostly)	use C interoperability to access IOR via a derived type
Fortran 90	native	slow	fast	IOR is copied into a F90 derived type
Fortran 77	functions	fast	slow	opaque object with access functions for each field
Python	native	fast	slow	IOR is presented as a Python object
Java	native	slow	fast	IOR is copied into a Java object

TABLE I
IMPLEMENTATION OF SIDL STRUCTS IN LANGUAGES SUPPORTED BY BABEL

Language	SIDL Arrays	R-arrays	
		Size-expression	Fixed-size
C	Pointer to IOR+access macros	Array pointer (*e. g.*, `int* a;`)	Embedded array (*e. g.*, `int a[42];`)
C++	Template+overloaded []-operator	Array pointer	Embedded array
Fortran 2003	Access functions	`bind(C)`-pointer (*e. g.*, `type(c_ptr);`)	C-Interoperable array (*e. g.*, `int (kind= sidl_int), dimension(1) :: a`)
Fortran 90	Native Fortran array or access functions—depending on data type		
Fortran 77	Opaque pointer with access functions		
Python	Numpy arrays [13] or generic sequence types		
Java	JNI array class wrapping the IOR		

TABLE II
IMPLEMENTATION OF ARRAYS AS STRUCT FIELDS IN BABEL

to a C++ `std::string` object, SIDL arrays are converted to the appropriate instantiations of the `sidl::array< >` template.

Contrary to the C binding, C++ servers come with an actual skeleton, which is used to convert C++ exception handling into SIDL exception variables. The client-side stub looks strikingly similar to the one used by the C binding, with additional code that converts any SIDL exceptions thrown by the invoked function to a C++ exception.

C. Fortran 77

Languages like Fortran 77 and Fortran 90/95 do not have the necessary compatibility with C. In Fortran 77, the struct is presented as an opaque integer parameter that comes with associated access functions for each field of the struct. While this notation is more verbose than, *e. g.*, the field access operator in C, it is still relatively cheap: the only performance penalty is a function call per field access. The access functions are implemented in C and are automatically generated by Babel in a way such that they are callable from Fortran. They take care of converting IOR data types to their Fortran equivalents. Figure 5 shows an example of how a field access is performed in legacy Fortran programs.

Memory management can be tricky in Fortran 77. The Babel compiler uses tagged pointers to determine the ownership of memory. If the lowest significant bit of a pointer is set, then the associated memory is borrowed. This implementation detail is completely transparent to the user.

```
Client
b = isValid(example_obj,  date,  &exception);

"Stub"
inline sidl_bool cal_example_isValid (
  /* in */ cal_example self,
  /* in */ const struct cal_Date__data* date,
  /* out */ sidl_BaseInterface  *_ex) {
  sidl_bool _result;
  _result  = (*self->d_epv->f_isValid)(self,date,_ex);

Server
sidl_bool cal_example_isValid (
  /* in */ cal_example self,
  /* in */ const struct cal_Date__data* date,
  /* out */ sidl_BaseInterface  *exception) {
  *exception = 0;
  {
    /* User implementation */
  }
}

  return  _result;
}
```

Fig. 3. Near-zero overhead call with a struct argument from C into C

```
namespace cal {
 /**
  * Symbol "cal.Date" (version 1.0)
  */
 struct Date : cal_Date__data {
  Date();
  ~Date();
  Date(const ::cal::Date &src);
  Date& operator =(const ::cal::Date &rhs);
  void _destroy();
  ::sidl::BaseClass  get_d_object()  const;
  void set_d_object(const ::sidl::BaseClass  &val);
  ::sidl::BaseInterface  get_d_interface();
  void set_d_interface(const ::sidl::BaseInterface  &val);

  ::sidl::int  get_month() const;
  ::sidl::int  get_day()  const;
  ::sidl::int  get_year()  const;

  void set_month(const ::sidl::int &val);
  void set_day(const ::sidl::int &val);
  void set_year(const ::sidl::int &val);

  // Assume that the object takes ownership of all
  // data and references previous owned by the rhs.
  ::cal::Date&  operator =(const struct s_Date__data &rhs);

  // Assume that the dest is uninitialized on entry.
  // Copy from this object into dest.
  void toIOR(struct s_Date__data &dest) const;

  void serialize(::sidl::io::Serializer  &pipe,
    const ::std::string  &name, const bool copyArg) const;

  void deserialize(::sidl::io::Deserializer  &pipe,
    const ::std::string  &name, const bool copyArg);
 }
}
```

Fig. 4. Babel-generated C++ interface for the struct in Fig. 1

```
subroutine processDate(d)
C       inout cal.Date d
        integer*8 d
        int*8 month

        call cal_Date_get_month_f(d, month)
C       do something ...
```

Fig. 5. Opaque pointers and access functions in Fortran 77

There is no difference between SIDL arrays and *r-arrays* in Fortran 77. Both are accessed through the same function interface.

D. Fortran 90

In contrast to Fortran 77, Fortran 90 introduces a native representation for structs: Fortran derived types. The resulting interface is very clean; the downside is that the skeleton needs to copy the IOR C-struct into the binary-incompatible Fortran derived type. This means that while access to the fields is cheap, passing a struct to a function always involves a copy operation. The skeleton implementation actually uses two indirections: the "regular" skeleton (in C) takes care of converting IOR to Fortran 90 data types. It then passes each (converted) field as an argument to the second part of the skeleton, which is implemented in Fortran 90 and copies all the arguments into a Fortran derived type which is the passed to the actual server-side implementation. This also works for structs nested inside of other structs.

SIDL arrays are generally passed as an opaque pointer with getter/setter functions. The names of these functions are considerably shorter than their Fortran 77 equivalents because they are declared as module procedures which has the effect of overloading a generic name as in `val = get(array, i, j)` and have the compiler decide which function to invoke based on the arguments. SIDL arrays of numeric types are wrapped into a derived type containing an opaque pointer to the IOR and a Fortran pointer to the array's raw data. Since data structure interoperability with C is not standardized, Babel uses `libchasm` [14] to generate an array adhering to the Fortran-vendor's specific data layout. In Fortran 90, *r-arrays* are always wrapped into SIDL arrays; there is no user-visible difference between the two.

Another peculiarity of the Fortran 90 binding is the generation of *type modules*. Due to limitations of the language it is necessary to split some derived type declarations (such as the declaration of the Fortran equivalent of a SIDL class) into a separate module. Only this way can circular dependencies be avoided, which would occur in situations where SIDL classes or interfaces are passed as method arguments. Apart from the additional file being generated, this has no practical effect for the user.

E. Fortran 2003

Fortran 2003 finally adds C interoperability via the `bind(C)` intrinsic module. Babel uses this feature to declared as structs as `bind(c)`-attributed derived types. This eliminates all of

```
module cal_Date
  use, intrinsic :: iso_c_binding
  use sidl
  type, bind(c) ::  cal_Date_t
    integer(c_int32_t) :: month
    integer(c_int32_t) :: day
    integer(c_int32_t) :: year
  end type cal_Date_t
  private :: get_month_p
  interface get_month
    module procedure get_month_p
  end interface
  private :: set_month_p
  interface set_month
    module procedure set_month_p
  end interface
  ...
end module cal_Date
```

Fig. 6. Fortran 2003 `bind(C)`-interoperability with C structs

the copying operations that were necessary in the Fortran 77 and Fortran 90 language bindings. An example is shown in Figure 6. This combines the performance of direct access with the convenience of a native data type. Since some data types (such as Boolean and Character types) are still not binary compatible, access functions are still generated, but they need only be used for these specific types. In contrast to the older Fortran versions, the Babel compiler generates skeletons for Fortran 2003 servers directly in Fortran instead of in C. Because the Fortran 2003 language only allows interoperable functions to return scalar values [15], the Babel compiler generates for these functions an additional wrapper of the Fortran skeleton in C, which converts an `out`-parameter to the return value.

F. Python

In Python, a C extension module for a Python object resembling the struct is generated. The extension module translates each access to a member of the Python object to an access of the corresponding field in the underlying IOR. The C extension also converts Python objects to the IOR. It is, for instance, possible to assign a Python list to an array field in a struct:

```
myStruct.doubleArray = [  1.0,  2.0,  3.0  ]
```

or we can even write

```
myStruct.objectArray  = [  sidl.BaseClass.BaseClass()    ]
```

In this example, the struct `objectArray` is an array of SIDL objects that is a field of the struct `s`. We are assigning a new instance of the generic SIDL base class.

The skeleton performs the necessary type conversions, acquires the Python interpreter's *global interpreter lock* (GIL) and starts the interpretation of the server code. By convention, Babel expects server implementations to return a tuple of return value and all `out`-attributed parameters. Upon completion, the skeleton copies the elements of the return tuple back into their corresponding `out`-parameters. It also handles the conversion of possible Python exceptions into their SIDL counterparts. The C extension module takes care of object (de-)serialization and of translating python field accesses into the appropriate actions on the IOR.

```
package cal;

public class Date {
  public int month;
  public int day;
  public int year;

  /** This is the holder inner class for inout
   * and out arguments for type <code>Date</code>. */
  public static class Holder {
    private cal.Date d_obj;

    /** Create a holder class
     * with an empty holdee object. */
    public Holder() { d_obj = null; }

    /** Create a holder with the specified object. */
    public Holder(cal.Date obj) { d_obj = obj; }

    /** Set the value of the holdee object. */
    public void set(s.Date obj) { d_obj = obj; }

    /** Get the value of the holdee object. */
    public s.Date get() { return d_obj; }
  }

  public Date() { }
  public Date(int month, int day, int year) { ... }

  ...
```

Fig. 7. Java representation of a SIDL struct

```
// SIDL definition
void advanceDate(inout cal.Date date);


cal.Date  d = new cal.Date(01,  19,  2038);
cal.Date.Holder  _d = new cal.Date.Holder(d);
try {
  advanceDate(_d); // modifies the inout argument
} catch (EpochFail ex) {}
```

Fig. 8. Using a *Holder* class instead of pointers

There is no difference between SIDL arrays and *r-arrays* in Python, but for numeric data types the Babel compiler uses the more efficient Numpy arrays [13] instead of regular Python sequence types.

G. Java

The Java binding uses an approach similar to the Python binding: A copy or reference to the IOR is used to create a Java object using the Java Native Interface (JNI) [16]. This makes passing a struct to/from a Java method more expensive but makes field access cheap because it does not go through the JNI. The stub uses the JNI to convert between the IOR and an object residing in the Java Virtual Machine (JVM).

As shown in Figure 7, a SIDL struct is represented as a Java class with the Java counterparts of all the IOR-struct's fields as public members. The class also contains a public inner *Holder* class used for `out` and `inout` arguments: Since Java does not support pointers, this class can be used to "hold" the struct in these cases. A code example is shown in Figure 8.

H. Remote method invocation

As indicated in Section II, Babel supports transparent remote method invocation [12]. A remote method call involves serial-

izing the arguments into a byte-stream, which is transmitted over the network. On the server side, the data is unpacked and the method implementation is invoked. Structs are serialized by packing all fields in a first-to-last, left-to-right order. Structs are essentially fixed-shape trees. A struct nested inside of another struct is serialized in-place by calling its respective serialization function. The server side performs the exact opposite actions to unpack the byte-stream again.

V. Experimental Evaluation and Discussion

To illustrate the performance considerations pointed out in Section IV, we ran benchmarks with structs of different sizes and with different data types. The different instances of the benchmarks were automatically generated from a language independent intermediate representation with the help of the BRAID code generator[3]. For each of the data types (`bool`, `float` and `string`) we generated SIDL definitions for structs containing $1 \ldots 128$ fields of that types.

The first two benchmarks consist of passing a struct to a no-op function ("call") and passing it ("access") to a function that accepts a struct $A = \{a_0, \ldots, a_n\}$ as `in`-argument and returns the field-reversed $A' = \{a_n, \ldots, a_0\}$ in an `out`-argument; an operation that is possible with all data types. In the beginning, the struct fields are initialized to `true`, i, and to a 16-character string filled with 13 spaces and i printed to 3 digits, respectively.

The third benchmark "bsort" shows what happens if there are many ($O(n^2)$) field accesses in the server function. This benchmark takes a struct of n integer fields as `in`-argument and returns a sorted struct as `out`-argument. The sorting algorithm is a naive bubble sort which has a quadratic worst-case behavior. The input is always reversed-sorted, and including the copying operation from input argument to output argument this results in a total of $2n + n^2$ field accesses.

The client implementation in all the benchmarks is always written in C. Since C always has the least overhead involved, this ensures a fair comparison of the different Babel language bindings.

The plots in Figures 9, 10, 11 and 12 show the number of instructions executed on a `x86-64` machine[4]. This number was measured by querying the `instructions`-performance counter provided by the `perf` [17] interface of Linux 2.6.32. In order to eliminate the instructions used for start-up and initialization, the instruction count of one execution of the benchmark program with one iteration was subtracted from that of the median of ten runs with 10^6+1 iterations each. The result was divided by 10^6 and plotted into the graph. The plots are logarithmic in both axes. The x-axis denotes the number of fields in the struct. The y-axis shows the number of instructions executed by the benchmark (lower values are better).

The benchmarks reflect many of the considerations put forward in the previous section:

- C is the fastest of the implementations, since it operates directly on the bare IOR.
- In C++ a constant overhead[5] has to be paid due to the way the language binding is implemented: The method dispatch mechanism goes through a wrapper function that encapsulates the called method in a `try`/`catch`-block, where possible C++ exceptions are translated into SIDL exceptions.
- Thanks to the C interoperability, Fortran 2003 is—for most data types—also offset from C only by a constant amount. One exception is the `bool`/access benchmark which uses the getter/setter function because of the incompatible binary representation of truth values between the two languages. In the Fortran 2003 case, the overhead is not paid for exception handling but for casting C pointers to their Fortran counterparts, transparently performed by the skeleton wrapper function.
- Fortran 77 has a low function call overhead, but a high cost for field accesses. In the "call"-testcases it is even faster than Fortran 2003, but the cost for the field access (*cf.* the "access" benchmarks) is higher because of the additional function call.
- The copy operation performed by the Fortran 90 implementation makes it stand out in all the "call"-testcases. Although this is obscured by the log/log scale of the plot, the overhead is actually linear (as one would expect from a copy operation). The overhead can be neglected if all the struct fields are accessed, as can be seen in the "access"-benchmarks.
- Python and Java incur the most overhead. In Java, field access is considerably cheaper as in Python, but the function call overhead is higher. Function calls in Java are expensive because of the conversion of the arguments from IOR to JNI objects. For higher[6] workloads, however, the just-in-time-compiled Java version quickly overtakes the interpreted Python implementation.
- The "bsort" benchmark shows what happens when there are many field accesses: This benchmark makes it clear that the copy overhead incurred by the Fortran 90 implementation (the skeleton copies the IOR into a native derived type) becomes negligible when there is a lot of work happening in the function. Particularly interesting is also the performance of the Java language binding, which shows that asymptotic behavior of Java is closer to the statically compiles languages, when the workload becomes significant.

Because the conversions from and to IOR often involve copying operations, strings are more expensive than other data types. This is reflected by the benchmarks in Figure 11: For all languages other than C and C++, the copy operation dominates the instruction count, making them virtually indistinguishable in performance. Comparing the "call" with the "access" testcase shows that Fortran 2003 does not copy the strings unless

[3] http://compose-hpc.sourceforge.net

[4] The test machine was an Intel Xeon E5540 running at 2.53GHz, with 8 threads and 6GiB of main memory running Ubuntu 10.04. The tests were compiled with the C, C++ and Fortran compilers of GCC 4.5.1 using standard optimization settings (`-O2`). The Python version was 2.6.5 and we used the SUN HotSpot 64-Bit Server version 1.6.0.22.

[5] ~ 30 instructions in the `bool` benchmark

[6] In our benchmarks the intersection is around $n \sim 10$.

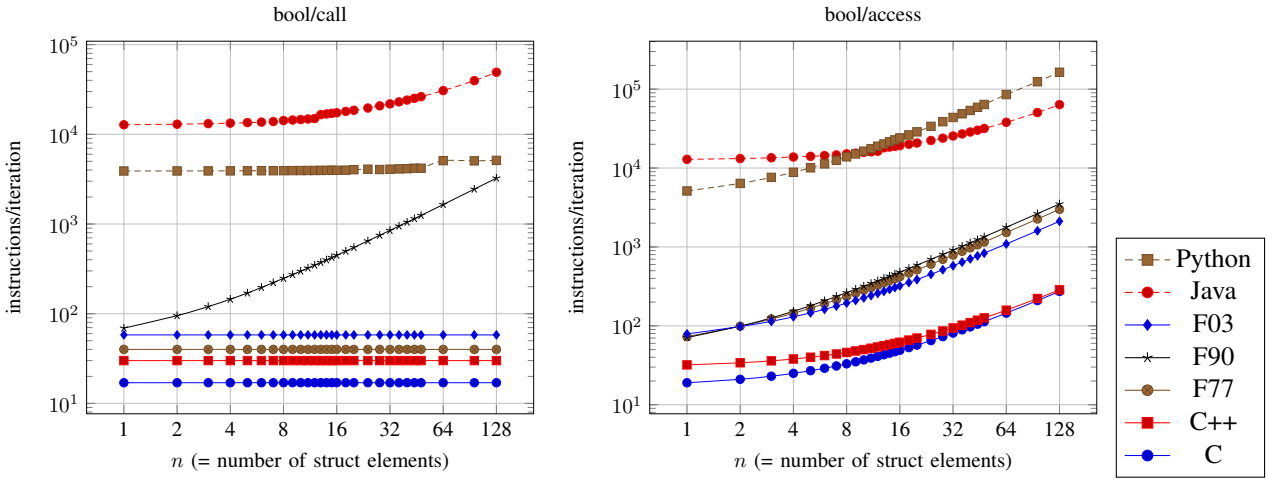

Fig. 9. Passing and accessing a struct of n booleans

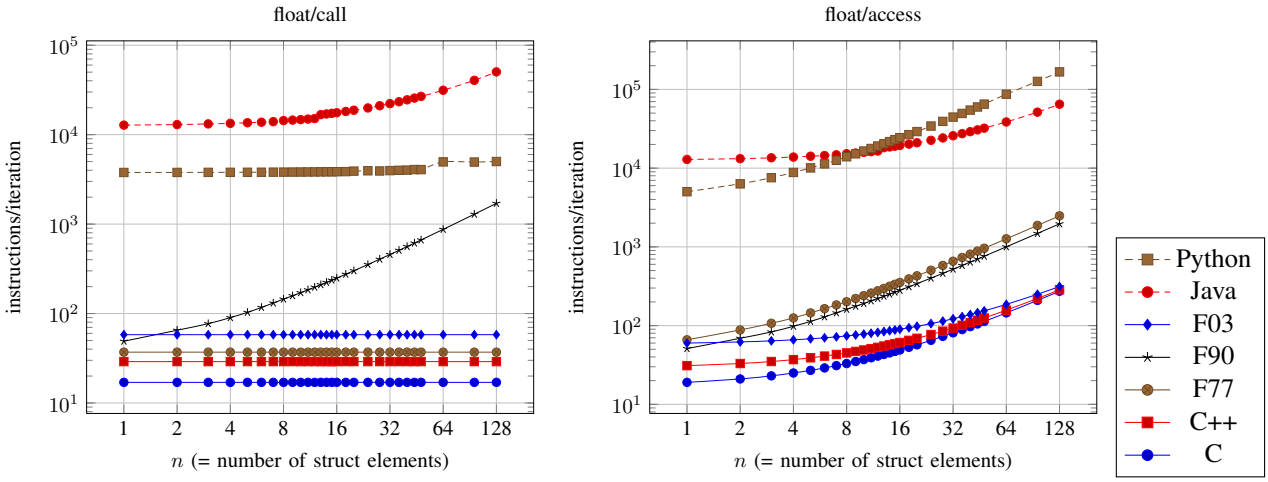

Fig. 10. Passing and accessing a struct of n floats

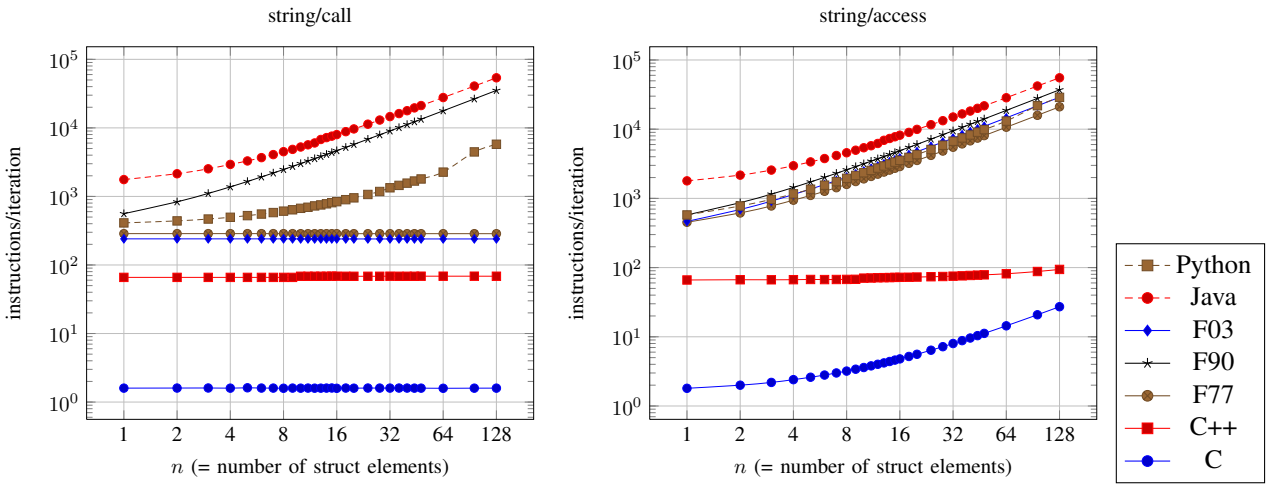

Fig. 11. Passing and accessing a struct of n strings

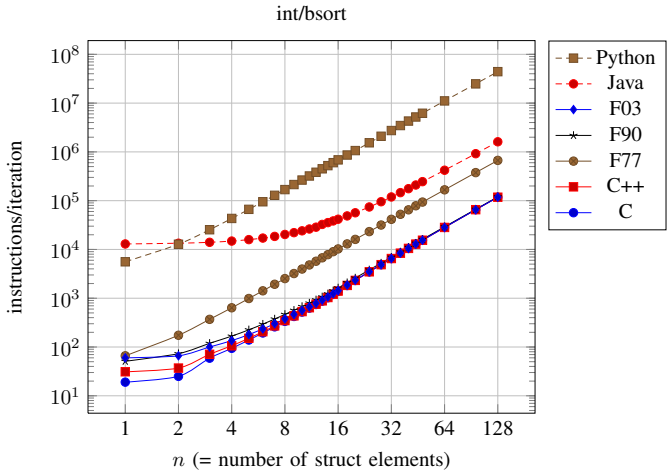

Fig. 12. Quadratic number of accesses of n integers

accessed.

In Section I we argued that because of the direct field access, structs offer a better performance compared to (SIDL) classes. The performance of the Fortran 77 language binding with its getter/setter function interface is also a lower bound for the performance of a class. In a class each member access would have to be another Babel method call.

A. Code size considerations

Scientific computing interfaces are often bound to specific mathematical models that render software engineering principles such as encapsulation (data hiding) counterproductive. By using a struct instead of a SIDL class, the interface is directly exposed and accessible and not hidden behind a pointless access function that does little more than wrapping class members.

Compared to using SIDL classes, structs can therefore significantly reduce the amount of glue code in an application; glue code that would have to be written by the user. As a consequence, they effectively reduce code size in most languages.

Structs are also a little more memory efficient. A SIDL class must also carry meta information such as pointers to its EPV and the EPV of each parent class. A struct occupies exactly the space occupied by its fields.

VI. A CASE STUDY: FLUXGRID

To demonstrate the real-world practicability of this feature we are now reporting our experience with a physics code from Los Alamos National Laboratory which was wrapped into a Babel component by Tech-X corporation [18], [19]. The program describes itself as follows [20]:

> `fluxgrid` is a code for reading in output from Grad-Shafranov equilibrium solvers and producing useful output for other codes. There are interfaces to a large number of commonly-used equilibrium codes, both direct and indirect, and adding interfaces to new codes is usually very simple. It currently can be used as either a standalone code or as a library which can be embedded into other codes such as `nimset`.

This code is a very typical example for the type program that is being componentized via Babel: It consists of about twenty Fortran 90 modules which are connected via a function interface that uses Fortran derived types (structs) to exchange data between the modules. The SIDL file, which is too long for inclusion in this paper, contains 15 different struct definitions; some of them nested. The median number of struct fields is 19; the largest struct counts 53 fields, the smallest only four. Combined, the structs contain 105 (SIDL) arrays and 8 other structs (which are also defined in the same file). The arrays have a dimensionality of up to 5. The most common base types for struct fields are int, double and string. To aid the task of defining the interface, the developers at Tech-X crafted a Python script that parses the Fortran sources and automatically generates the SIDL file with all the derived types used by `fluxgrid`'s interfaces.

An external constraint of the design was that the resulting code must compile with a selection of legacy compilers. The "babelized" version therefore uses the Fortran 90 binding instead of the more efficient Fortran 2003 binding. Since the majority of the data is encapsulated inside of SIDL arrays (*cf.* Section IV-D) the overhead is still acceptable. Another design decision was that the existing Fortran 90 sources were not to be modified in the process. For this reason, an additional layer of Fortran 90 glue code was added, which translates the derived types from the Babel method arguments into the derived types used by the original Fortran implementation. With the Fortran 2003 language binding, the SIDL arrays could be replaced by *r-arrays* and this copy operation could have been eliminated.

Our measurements include this additional overhead. Using the `geqdsk` input set provided by Tech-X, we measured a 1.3% overhead for calling the Babel version of `fluxgrid` from a driver written in C++; whereas the original version is driven by a Fortran program that calls the library functions directly.

This example shows that the Babel struct extension was successfully used in practice to wrap existing code into a well-defined component interface, while retaining the original data layout[7] for input and output. It is now possible to orchestrate the Fortran 90 core directly from C++ and Python which enables a much tighter coupling between components written in different programming languages than previously possible.

VII. Outlook and Future Work

With the addition of struct data types, Babel comes one step closer to providing a full programming ecosystem between multiple languages. Using classes to exchange data is oftentimes an overkill; structs allow users to write more poignant and compact code that will also have higher performance. Babel's struct support degrades gracefully (to auto-generated classes or function interfaces) for languages which do not support such a feature natively. The highest performance is achieved by the C, C++ and Fortran 2003 language bindings: They provide *zero-copy*, *direct access* struct implementations that are set off only by a constant factor in most cases (comparable to a native call in C).

With this feature in Babel we provide the computational scientist with another (and much requested) choice for the data structures to use for their interfaces. The measurements included in this paper show the detailed performance trade-offs for different data types and programming languages. We hope for this work to make choosing the most appropriate representation for a specific domain a little easier.

Acknowledgments: We would like to thank Scott Kruger and Roopa Pundaleeka from Tech-X Corporation for providing us with their version of `fluxgrid` and for sharing their experiences with using the Babel extensions presented in this paper.

References

[1] T. Dahlgren, T. Epperly, G. Kumfert, and J. Leek, *Babel User's Guide*, Lawrence Livermore National Laboratory, July 2006, version 1.0.0.

[2] *The Common Object Request Broker: Architecture and Specification*, Object Management Group, February 1998. [Online]. Available: http://www.omg.org/corba

[3] "CORBA website," http://www.corba.org, Object Management Group.

[4] N. Brown and C. Kindel, "Distributed component object model protocol-DCOM/1.0," Microsoft Corporation, Redmond, WA, Tech. Rep., November 1996.

[5] Microsoft Corporation, ".NET homepage," http://www.microsoft.com/net.

[6] Sun Microsystems, "Enterprise JavaBeans downloads and specifications," http://java.sun.com/products/ejb/docs.html, 2004.

[7] D. E. Bernholdt, W. R. Elwasif, J. A. Kohl, and T. Epperly, "A component architecture for high-performance computing," in *Proceedings of the Workshop on Performance Optimization via High-Level Languages (POHLL-02)*, New York, NY, June 2002.

[8] R. Sessions, *COM and DCOM: Microsoft's vision for distributed objects*. Wiley and Sons, 1998.

[9] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren, "The CCA component model for high-performance computing," *Intl. J. of Concurrency and Comput.: Practice and Experience*, vol. 18, no. 2, 2006.

[7] This is modulo the array layout, which had to be sacrificed to support the legacy Fortran 90 compiler.

[10] G. Kumfert and J. Leek, "Writing a protocol to support RMI," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-TR-220292, February 2006.

[11] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (WSDL) 1.1," 2001, http://www.w3.org/TR/wsdl.

[12] G. Kumfert, J. Leek, and T. Epperly, "Babel remote method invocation," in *Proc. 21st Int'l Par. Dist. Proc. Symp. (IPDPS'07)*, D. K. Panda, Ed. IEEE Computer Society, March 2007, to appear.

[13] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," http://www.scipy.org/, 2001–.

[14] C. E. Rasmussen, M. J. Sottile, S. Shende, and A. D. Malony, "Bridging the language gap in scientific computing: the chasm approach," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 2, pp. 151–162, 2006.

[15] J. C. Adams, W. S. Brainerd, R. A. Hendrickson, R. E. Maine, J. T. Martin, and B. T. Smith, *The Fortran 2003 Handbook. The Complete Syntax, Features and Procedures*, 1st ed. Springer, 2009.

[16] S. Liang, *Java Native Interface: Programmer's Guide and Reference*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] A. C. de Melo, "The new linux 'perf' tools," Slides from Linux Kongress, 2010, http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf.

[18] A. H. Glasser, C. R. Sovinec, R. A. Nebel, T. A. Gianakon, S. J. Plimpton, M. S. Chu, D. D. Schnack, and the NIMROD Team, "The nimrod code: a new approach to numerical plasma physics," *Plasma Physics and Controlled Fusion*, vol. 41, no. 3A, p. A747, 1999. [Online]. Available: http://stacks.iop.org/0741-3335/41/i=3A/a=067

[19] A. L. Rosenberg, D. A. Gates, A. Pletzer, J. E. Menard, S. E. Kruger, C. C. Hegna, F. Paoletti, and S. Sabbagh, "Modeling of neoclassical tearing mode stability for generalized toroidal geometry," *Physics of Plasmas*, vol. 9, no. 4567, 2002.

[20] A. Glasser, T. Gianakon, and S. Kruger, "`fluxgrid` source code," 2011.