Fast Native Function Calls for the Babel Language Interoperability Framework

Dietmar Ebner Thomas G. W. Epperly
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California
{ebner|epperly2}@llnl.gov

***Abstract*—Babel is an open-source language interoperability framework tailored to the needs of high-performance scientific computing. Its primary focus is on fast in-process communication across various languages. In doing so, some additional call overhead is often inevitable. For several pairs of languages, however, shortcuts exist that allow for more efficient function calls. As Babel is a dynamic framework, the particular set of languages involved is often only known at runtime.**

In this work, we present a simple yet very effective optimization that can be used to reduce the call overhead between various pairs of languages. In particular, our optimization is applicable if caller and callee are implemented in the same language. We implement and evaluate these techniques for C++ and Python. When applicable, our optimization virtually eliminates the overhead for a small memory cost. Compared to previous versions of Babel, this means a speedup ranging from about 5x for simple numerical argument types up to roughly 125x for strings.

I. Introduction

Supercomputing simulations are critical for global warming predictions, energy research, and for advancing basic scientific understanding. With supercomputers becoming more powerful in each generation, more sophisticated and detailed simulations become feasible. Also, these simulations often integrate mathematical models from different domains to achieve better precision, *e. g.*, climate models might be combined with social models to predict emissions of carbon dioxide. This trend has led to large and complex systems that are difficult to maintain. One of the main reasons for this complexity is legacy code that often cannot be replaced for technical or economical reasons.

One approach 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 [1], [2], Microsoft's (D)COM [3] and .Net [4], or Sun's JavaBeans [5]. The Common Component Architecture (CCA) [6] 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 performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-440073

A major requirement for scientific computing is language interoperability. This allows the component paradigm to incorporate legacy software written in mixed languages. Babel [7] is focused on the special needs of high-performance scientific computing. As such, it can be used stand-alone or as part of the full CCA [8] component framework.

In order to address interoperability challenges, Babel makes use of the scientific interface definition language (SIDL). SIDL builds on previous work such as CORBA [1] or COM [9] by tailoring the idea to the needs of scientific computing. This includes support for dynamic multi-dimensional arrays, array strides, single and double precision complex numbers, and structs.

Babel transparently supports both fast in-process function calls and remote method invocation (RMI) [10]. In the latter case, caller and callee may reside in a different address space or on different machines. Babel does not impose constraints or assumptions on the parallel communication model used within components, which may be any combination of MPI [11], OpenMP [12], Posix Threads [13], Global Arrays [14], or similar techniques. However, Babel allows for composition of parallel components, *e. g.*, the Co-op project [15], [16] found that an RMI paradigm for MPMD programming was easily understood and very effective for application developers [17], [18], [19], [20]. The domains in which Babel is used are widespread and range from applications in chemistry, astronomy, and biology to mathematical solvers, programming models, and performance monitoring tools. Established users of the Babel language interoperability middleware and/or the CCA as a whole include the *hypre* preconditioner library [21], the ComPASS project [22], the CSDMS project [23], the FACETS project [24], and the MPQC quantum chemistry package [25], [26], [27].

Based on SIDL interface specifications, Babel assists the developer by generating language-specific prototypes. It also generates the necessary glue code for non-native method invocations. For each of the supported languages,

U.S. Government work not protected by U.S. copyright

Babel tries to make data passed into or returned from functions appear as "natural" as possible, *e. g.*, a string appears as a character array in C while it is represented as a `java.lang.String` object in the Java bindings. Thus, each Babel call involves some additional overhead to do the necessary conversions, implement a common object model (even on top of procedural languages such as C or Fortran), and to provide a transparent exception mechanism. For non-local calls, there is an additional, much larger, overhead to marshal and un-marshal the data and to do the network transfer.

Motivation

Until recently, the main focus in Babel development has been correctness and broadening of its capabilities. In fact, most applications use a very coarse grained component model with infrequent Babel calls so that call overhead due to language interoperability hardly matters, *e. g.*, experiments for course-grained interface descriptions [28] clearly show that the overhead of Babel is well within measurement imprecision — usually below 1%.

However, there are also interesting applications where the overhead of method invocations can become a significant factor compared to the computational payload of a function. One of these examples is TSTT (Terascale Simulation Tools and Technologies), which aims to establish a community standard interface for mesh refinement codes. Performance studies [29] showed considerable overhead compared to a native interface, which led to the adoption of fast native interfaces instead of SIDL as their primary technique.

Also, for several users, language interoperability is not necessarily the main argument for adopting Babel. Instead, they are sometimes mainly interested in clean interface specifications, remote method invocation, or the object oriented programming model provided on top of traditional procedural languages such as Fortran. For these users, even a moderate overhead is often reason enough *(a)* not to adopt Babel at all or *(b)* design applications with these performance considerations in mind, often leading to less intuitive interfaces [26]. Thus, the primary motivation of this work is to reduce the Babel overhead for these users, thereby facilitating adoption and impacting a larger audience.

Performance optimizations for fine-grained interfaces are complicated by the fact that it is in general only known at runtime *(a)* if caller and callee reside in the same address space and *(b)* which pair of languages is actually involved in a particular method invocation. This is due to polymorphic function calls and Babel's support for transparent remote method invocations. In this work, we propose an in-process optimization that can effectively eliminate the call overhead for various pairs of languages for a small memory cost. In particular, this is useful when caller and callee are implemented in the same language, which is very common scenario in practice. The main contribution of this work is a dynamic optimization that avoids unnecessary conversions and copying whenever possible by generating efficient dynamic delegates in the auto-generated client-side glue code. This approach retains all the advantages of dynamic binding and remote method invocation while providing roughly the performance of native calls when both objects end up residing in the same address space and being implemented in the same language.

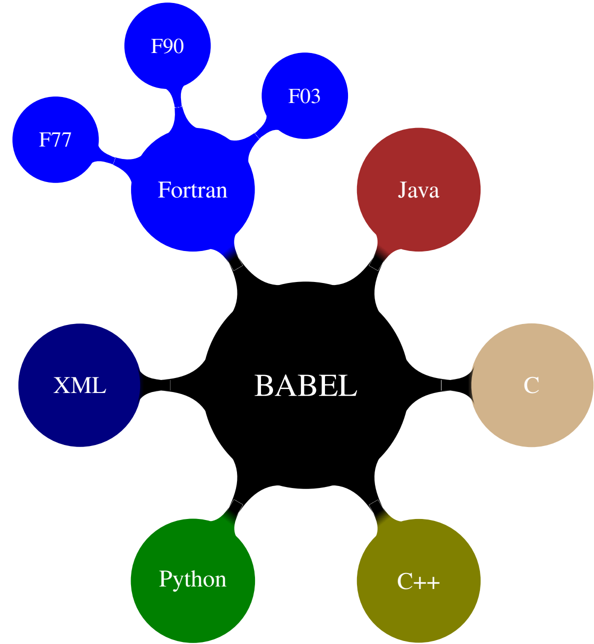

Figure 1. Supported programming languages.

We describe Babel's general approach to language interoperability in more detail in Section II. Our extensions and implementation details for C++ and Python are presented in Section III. A detailed computational evaluation showing the effectiveness of our approach can be found in Section IV.

II. Babel Architecture

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

Backends are available for a large and growing set of languages, cf. Figure 1. Restricting Babel to the least common denominator across all these languages would be a non-practical approach. Instead, Babel tries to take advantage of native language features such as builtin 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 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.

Figure 2 depicts the general scheme 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 entry point vector (EPV), and converts eventual 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. The overhead introduced by this scheme depends on the particular pair of languages and the type of the arguments. For remote objects, the only difference is that the EPV will not directly point to the skeleton but to a babel-generated function (remote method stub) that marshals and un-marshals the arguments and performs the necessary network transfer.

III. Approach

Babel supports virtual function calls even on top of procedural languages such as Fortran. Consequently, it cannot rely on builtin object-oriented language features. Instead, it implements its own virtual function table and generates the necessary dispatch code for the various supported languages in the client stubs. Dynamic dispatch using virtual function tables was first introduced in Simula [30] and is today the preferred technique for widely used languages such as C++ [31].

In Babel jargon, each object or interface carries a reference to an entry point vector (EPV) that defines the set of member functions supported by the corresponding type. Figure 3 shows the memory layout for a Babel object with a simple inheritance structure. Solid lines denote generalization while dashed lines stand for implementation of interfaces.

Babel maintains a strict separation between client and server code to ensure that components can be distributed in binary form together with the corresponding SIDL file. In the current design, the method entry points stored in the EPV are the only way to "cross" the barrier between client and server code. Skeletons always expect and return data in Babel's IOR and translate arguments to and return values from the particular implementation language.

This approach has many advantages. However, in terms of performance, it is often not optimal. For several pairs of languages, there are "shortcuts" that involve less runtime overhead than converting to and from Babel's IOR. The most obvious and also the most common case are method invocations where both client and server are implemented in the same language.

To motivate this with a concrete example, consider a string being passed using mode in-out from a C++ client to a member function of a SIDL component also implemented in C++. Babel's C++ bindings use a reference to a `std::string` object, which boils down to a simple address passed on the stack. In Babel's IOR, however, the same string is represented as a `char**` pointer, as implementations are allowed to return a string different from the one being passed. The stub calls `strdup` to allocate memory and clone the string wrapped by the `std::string` object. After the call is finished, the result string is assigned to the `std::string` object (which may involve memory reallocation) and the temporary allocated memory has to be released. On the server side, the skeleton has to wrap the raw character pointer using a fresh `std::string` object created on the stack, pass it to the actual implementation by reference, and copy the raw data back to the input string. Depending on the length of the string returned, this last step may involve dynamic memory reallocation as well. In practice, the whole procedure is roughly two orders of magnitude slower than a native call by reference.

We propose a solution that requires modification to Babel in two areas. The first is a language-independent generalization of Babel's IOR. The second are language-dependent extensions of the various backends, of which Python and C++ have been implemented so far. The latter is the more interesting example and is discussed in more detail in the following sections.

Generalized Dispatch Tables

Support for dynamic dispatch in Babel implies that we cannot statically deduce the actual implementation language used for a particular method invocation in general. Instead, it depends on the *dynamic* type of an object, *e. g.*, invoking method `b()` on an object reference of type `B` in the example introduced in Fig. 3 might be either a Python or a C++ call, depending on whether the actual object is of type `B` or `C`. Also, the amount and type of information we need to provide in order to implement efficient in-process method invocations strongly depends on the particular implementation language.

Our solution is an extension to Babel's EPV data structure that allows language backends to expose alternative method entry points with a calling convention different from Babel's IOR. For each method, the EPV is augmented by two

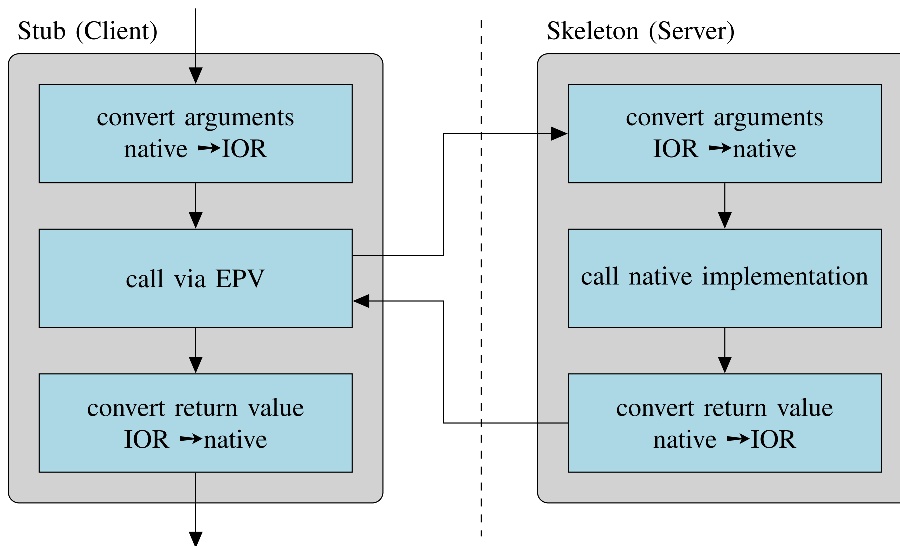

Figure 2. Babel method invocation. Arguments and return values are converted to Babel's intermediate object representation (IOR) before being passed.

additional fields, *i. e.*, an enumeration type and an opaque pointer. Method stubs evaluate the enum in order to determine the meaning of the opaque pointer, which might be anything from regular function pointers to language-specific handles or secondary data structures. This approach allows language backends to expose the information necessary to implement more efficient method invocations. However, a Babel backend is considered complete even if it does not implement these extensions.

Two modifications are necessary for language bindings to make use of these features. On the server side, Babel calls a generated function `set_epv` for each component to initialize the static EPV and, optionally, the additional fields introduced above. On the client side, additional dispatch code has to be generated that takes advantage of these language specific "shortcuts" if present.

Since EPVs are static information, both the additional memory and runtime overhead for initialization of dynamic dispatch tables is negligible in practice. However, for most language pairs, there is a small runtime overhead necessary to implement the dynamic dispatch.

C++ Bindings

Babel's C++ bindings closely model the method signatures and inheritance structure defined in the SIDL file. There is a one-to-one correspondence between basic SIDL types and their C++ equivalents. Whenever feasible, SIDL types are mapped to their native C++ or STL equivalents. For more complicated types such as arrays, Babel provides a runtime library with corresponding C++ interfaces. The semantics of the client side bindings are similar to C++ smart pointers, *i. e.*, reference counting is handled implicitly. Interfaces are implemented using abstract base classes. Native exception mechanisms are used to handle SIDL exceptions.

The set of classes generated by Babel for the class hierarchy introduced in Figure 3 is show in Figure 4. Both `BaseInterface` and `BaseClass` are part of Babel's runtime library. The user implements only the classes to the right of the dashed line. The remaining classes are glue code generated by Babel and basically wrap the IOR for the corresponding SIDL class.

Babel generates a static `_create()` member function that is used to create and initialize new SIDL objects. Initialization is performed recursively for base classes and records a pointer to a newly created implementation object in the `data` field of the IOR. The generated skeleton casts this field back to the appropriate type and invokes the user-defined implementation after converting in-arguments. Note, that the generated member functions for client stubs are non-virtual and can thus be bound at compile time as Babel implements virtual call semantics in a language-independent way.

Except for so-called cv-qualifiers (const, volatile), type signatures of Babel-generated method stubs and of user-supplied implementations are equivalent. Thus, in the case of C++ calling C++, we ideally "replace" the generated method stub with the actual implementation to avoid the usual overhead. In software engineering literature, this technique is usually referred to as *delegation*. The key to implement this efficiently are member function pointers — one of the darker corners of C++. We will thus continue with a little excursion to recap the idiosyncrasies of this language feature before explaining how we can employ it for zero-overhead native function calls in Babel.

C++ Method Function Pointers: Unlike regular function pointers, method function pointers in C++ are a relatively

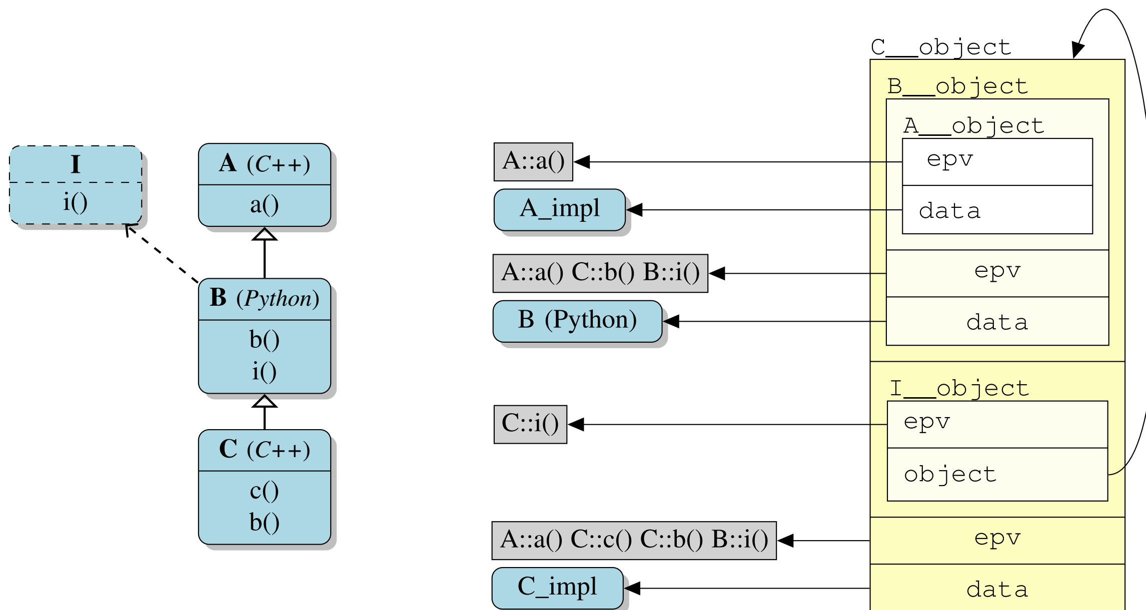

Figure 3. Babel object layout for a simple inheritance structure.

complex feature. As their name suggests, they are designed to hold the address of a C++ member function. The standard is very restrictive in what you can do with them: they can be set to `NULL`, compared for equality and inequality (as long as both operands are of the *same* type), and compared against zero. According to the C++ standard[1] [32], they can also be cast using `reinterpret_cast` to a member function pointer of an *unrelated* class. Unfortunately, this (a) does not work for several compilers in practice and (b) is not really useful as everything you can legally do with it is to cast it back to the original type.

To understand why, recall that for virtual functions, the actual function being called depends on the supplied `this` pointer. There is a vast diversity in how compilers implement this behavior. Some (more exotic) compilers emit a small piece of code (thunk) that performs necessary lookups and pointer adjustments before invoking the actual implementation. This technique is simple, but involves some runtime overhead due to the additional indirection. A more common technique employed in almost all major compilers is to add additional fields that specify the `this` pointer offset and, in the case of virtual methods, the index in the virtual function table. Thus, depending on the target machine architecture and the compiler being used, the size of member function pointers can be anywhere between four and 20 bytes. The size of member function pointers also depends on the particular inheritance structure as compilers apply different optimizations for single, multiple, and virtual inheritance in practice. Thus, casting a member function pointer to a different type can change its size and may lead to bizarre side effects.

Fast Delegates for C++ Client Bindings: Client bindings generated by Babel are strictly separated from the actual implementation and do not depend on whatever language has been used to implement a particular class. This also means that the user-defined type of the implementation class is not known to the compiler. In order to implement efficient function calls, we need a way to (a) communicate the proper `this` pointer and the method entry point between client and server and (b) reliably invoke the actual implementation.

Surprisingly, the last part is easy to achieve. The reason is that member function pointers can be declared and invoked on incomplete types, *e. g.*, the following code sequence is legal C++:

```
class A;
void invoke(A *obj, void (A::*ptr)()) {
  (obj->*ptr)();
}
```

The compiler has to be able to generate correct code knowing nothing about class `A`. The anonymous class basically

[1] Section 5.2 10/9

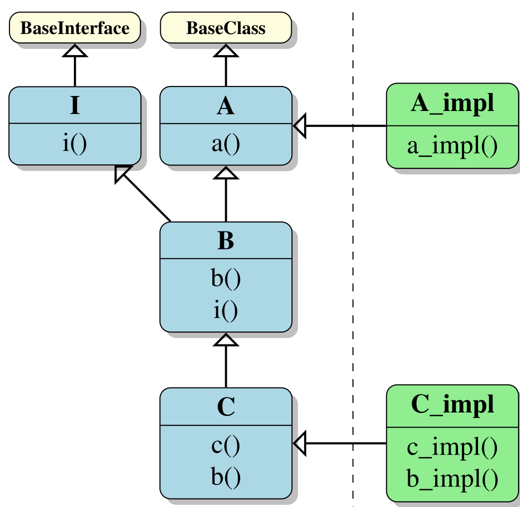

Figure 4. Generated C++ bindings for the class hierarchy introduced in Figure 3.

forces the compiler to disable all the optimizations it might do for a known inheritance structure. This leads to the bizarre situation that casting of member function pointers is standard-compliant but non-portable, their invocation on the other hand, once the cast succeeds, is non-standard but portable in practice. An elegant way to handle compilers not supporting unrelated casts is by employing partial template specialization based on the actual *size* of member function pointers. Interested readers are referred elsewhere [33] for an excellent in-depth discussion.

The one ingredient missing is the proper `this` pointer. For classes implemented in C++, Babel maintains such a reference in the data pointer shown in the object layout in Figure 3. Note, that there is one such pointer per class in the inheritance hierarchy and it depends on the *dynamic* type of the actual object which one to use. This information cannot be inferred by the generated client stub in general.

Our solution is to pass the offset relative to the beginning of the IOR along with the method function pointer on to the client. The opaque EPV pointer points to a struct that contains both the member function pointer and the proper offset to calculate the `this` pointer. For interfaces, the correct `this` pointer can be obtained by adding the offset to the `object` pointer indicating the beginning of the actual object; see Fig. 3 for reference. In the case of static functions, we can avoid this indirection and store the native function pointer directly in the generalized EPV.

On the client side, a small thunk is generated instead of the default stub code that dynamically chooses between the native version (if applicable) and the generic fallback code. The dispatch code dynamically checks if an object supports fast native calls by exposing the necessary method entry points. It then computes the `this` pointer, fetches the method function pointer, and directly invokes the user-code. Otherwise, control is handed over to the generic Babel stub.

For native function calls, this is always a win; see Section IV for computational results. Also, there is almost no associated memory cost apart from the slightly bigger EPVs and a few additional bytes in code size needed for the dispatch thunk. It does, however, add a few extra cycles to the costs of a function call if the check fails, *i. e.*, a non-native implementation is called.

We can do considerably better by spending a few extra bytes per method in order to cache the outcome of the dispatch process. The result is a pair consisting of a `this` pointer and a method function pointer, either pointing to the generic Babel stub or the native user code. The thunk generated for each method then basically looks like the following code sequence.

```
inline void foo(a_1, a_2, ... , a_n) {
  (this_foo->*mfp_foo)(a_1, a_2, ... , a_n);
}
```

A reasonably optimizing compiler will inline this code sequence, which brings the cost of a native Babel call down to the cost of a regular member function pointer invocation once the pointer is cached. For compilers without support for link-time optimization, it is essential to generate the method dispatch in the header file for the obvious reasons.

We implement both *eager* and *lazy* caching. In the first case, we bear the costs at object creation time in favor of faster method invocation. In the latter case, the member function pointer and the `this` pointer are computed at the first invocation, requiring an additional runtime check. However, this check is fully predictable and turned out to be very cheap on modern architectures as they execute code speculatively based on sophisticated branch prediction units.

For some compilers such as `gcc` we can still do slightly better. As discussed before, invoking a member function pointer can be quite costly as it may involve `this` pointer adjustments and lookups in the virtual dispatch table (vtable). For recent versions of `gcc`, things get even worse as the compiler applies a tricky optimization to decrease the size of member function pointers at the expense of a small runtime overhead: the vtable index and the method entry point share the same field in its internal data structure. Addresses are always aligned and thus even. For a vtable index i, the value $2i+1$ is stored, which is always odd and allows the compiler to distinguish among the two cases. The necessary dynamic check adds a few extra cycles to the costs of a method function pointer invocation.

Fortunately, `gcc` implements a nifty C++ language extension to make up for this. It allows developers to extract the actual function pointer that would be called for a given pair of object and method function pointer, *e. g.*, we can cast a method function pointer of type `void (A::*)(int)` to a regular function pointer of type `void (*)(A*, int)` by supplying a concrete `this` pointer. By employing this feature, the

space overhead for caching can be reduced to two machine words per member function (the `this` pointer and a more efficient regular function pointer) and the costs for native method invocations is reduced to the costs of an indirect function call — as low as we can get for polymorphic function calls.

There are two rather subtle differences in the programming model compared to regular Babel calls. First, the generic skeleton generated by Babel catches language specific C++ exceptions and converts them to generic SIDL exceptions before passing them on to the user. By applying our optimization, this conversion does not happen and the user will observe the unmodified C++ exception. This may change the behavior of existing code but is easy to handle in practice. The second difference is related to `out` arguments. In the traditional setting, Babel consistently initializes those arguments even if they are never defined in the user-supplied code. Again, this initialization does not happen in the optimized case. Code relying on this behavior has been always undefined but works reliably using older versions of Babel or when the optimization is disabled.

Python Bindings

As for C++, namespaces and object hierarchies defined in SIDL map nicely to native Python packages and modules. Babel uses Python C extension modules that wrap the internal object representation and implement argument marshaling and method dispatch.

As Python is a dynamic language, most of the difficulties of invoking member functions of incomplete types discussed before do not apply. Instead, Python provides bound member functions that "remember" their context. Depending on whether or not a method is ultimately implemented in Python, a callable bound to the actual implementation object or the regular stub is returned upon attribute lookup. As for C++, a reference to an object of the user-supplied implementation class is maintained in the data pointer, cf. Figure 3. Its computation involves the same steps as discussed before in the context of C++.

Depending on whether or not a method is declared `static`, we apply a different method to implement the attribute lookup.

- Static methods are resolved at module initialization time and override the default client stub in the module dictionary.
- For non-static member functions, we override Python's default implementation for `tp_getattro` that is called to lookup object attributes dynamically. For native member functions, this procedure returns a callable bound to the actual implementation. Otherwise, we defer the lookup to `PyObject_GenericGetAttr`. This process is less efficient but does not require the maintenance of a dictionary per object. However, different trade-offs are possible.

IV. Experimental Evaluation and Discussion

In this section, we present a performance evaluation of the optimization presented in this paper. The results are very encouraging, showing call overheads practically on par with native virtual function calls in C++ for a very moderate memory overhead while retaining all the flexibility of regular Babel objects.

Measuring fine-grained call overheads can be a challenging problem. We thus subsequently discuss the performance measurement techniques employed to minimize external interference factors before presenting detailed experimental results for various data types on a recent Linux kernel using the two major compilers used in the scientific community: Intel's `icc` and the GNU compiler `gcc`.

A. Methodology

Estimating performance on modern x86 microarchitectures has become an almost impossible task. The reasons are manifold: hyper-threading, programmable decoding units, branch prediction, out-of-order execution and aggressive speculation are only some of the causes. Additional noise is generated by cache coherence protocols, deep memory hierarchies, shared systems resources such as buses, and operating system overhead.

The costs for Python method invocations have been measured using the builtin `timeit` module with 10.000 iterations and a repeat count of three. We measure the duration of a method invocation for an empty function using various argument types. The Babel version is a modified prerelease for 1.5.

Reliable results for C++ are much harder to obtain. We use CPU cycle counters (`rdtsc`) to obtain cycle-accurate timings. Calls are executed repeatedly (10^6 times). We report the average call costs using statically linked binaries. The duration of an equivalent empty loop is used to account for the loop overhead. An empty volatile-qualified inline assembler block was used to prevent the compiler from optimizing the loop away. As we execute calls repeatedly, we deliberately measure best-case performance as caches and branch target buffers will perform near-optimal. In practice, calls may involve a higher overhead.

Operating system overhead can lead to considerable noise ($>\pm 10\%$). We thus ran our C++ benchmarks directly in the context of the kernel and disabled software interrupts before beginning the measurements (`cli`). A serializing instruction (`cpuid`) is issued to force completion of in-flight instructions executed out of order. Furthermore, we disabled power management, symmetric multi-processor support (SMP), and hyper-threading. Thus, apart from non-maskable interrupts, we can be certain that no other code is interfering with our profiling runs. We use a 32 bit stock Linux 2.6.32 kernel with a corresponding Kernel Mode Linux[2] (KML)

[2]http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml

Argument Type	Native	Babel 1.4	Caching		
			none	eager	lazy
Array	6.8	86.7	33.0	6.0	7.0
Struct	21.0	40.0	40.9	20.5	20.1
String	7.4	874.2	32.9	6.2	7.0
Complex	8.0	69.7	34.6	6.0	7.0
Int	6.4	38.3	31.2	5.5	7.5
None	6.0	37.2	30.5	5.4	5.6

Table I
AVERAGE CALL OVERHEAD IN MACHINE CYCLES FOR C++ COMPARING THE COSTS FOR NATIVE METHOD INVOCATIONS AND CALLS THROUGH THE BABEL MIDDLEWARE USING GCC.

	no caching	eager caching	lazy caching
memory (static)	12	12	12
memory (per obj.)	0	12/8	12/8
caching	none	object creation	first invocation

Table II
OVERVIEW OF VARIOUS CACHING STRATEGIES. MEMORY OVERHEAD IS REPORTED IN BYTES PER METHOD FOR `icc` AND `gcc` RESPECTIVELY (32 BIT LINUX).

patch to execute regular processes in ring zero. Results are shown both for `gcc` version 4.3.2 and the Intel compiler in version 11.1. Benchmarks were executed on a 2.4 GHz Intel Core™ 2 Duo system with 2GB of DDR2 memory. Results can vary significantly for different compiler flags. To make for a fair comparison, we used the highest particular optimization level (`-O3`) and enabled machine-specific code generation (`-mtune=native`).

For most simple data types, our results are very precise showing almost no variability. However, more complicated types such as strings or arrays involve temporary memory allocation and copying. For these benchmarks, some variability caused by the memory subsystem is unavoidable. We thus repeat measurements 100 times, always disregarding the first run. For all data points, the sample standard deviation is smaller than 10%. Simple cases such as integer arguments show almost no deviation.

B. Computational Results

Table I shows experimental results for C++ calls with various argument types in machine cycles. For space limitations, we only present interesting data types. Other basic data types such as floating point numbers, chars, or enums roughly show the same behavior as integer values. Arguments are passed by value (mode `in` in Babel jargon). The first data column shows the costs of native C++ virtual method invocations. Column "Babel 1.4" shows results for Babel without the optimizations proposed in this paper, everything else being equal. Data for the various caching strategies discussed in Section III can be found in the last three columns ("Caching).

Data shown in Table I has been gathered using `gcc`, which provides a C++ language extension that allows for more efficient method pointer invocations. For eager and lazy caching, the Intel compiler produces only slightly worse results but requires more memory, cf. Table II. However, `icc` produces slightly more efficient code without caching, *e. g.*, an integer method invocation costs about 25 instead of 30.5 cycles.

It is interesting to see how our results compare to those published about eight years ago for Babel 0.7 [6] using older hardware (500MHz Pentium™ III) and compiler technology. In terms of machine cycles, we observe an astonishing improvement both for C and C++ by a factor of about 2.9 and 2.7 respectively. This is also true for simple data types where the potential for compiler optimizations is quite limited. Thus, a large fraction of this is probably due to advances in processor technology, *e. g.*, branch prediction and superscalar micro-architectures. The speedup in hardware is even more impressive as clock rates are up by roughly another factor of five on modern machines.

In the case of C, the Babel overhead compared to the costs of a native function call is moderate. This is not surprising as Babel's intermediate representation (IOR) is written in C and hardly requires additional transformation. Two noteworthy exceptions are raw arrays (`rarrays`) and arrays with order specifications. In the first case, raw data pointers can be used instead of SIDL arrays to resemble existing interfaces. However, in the current implementation, Babel internally wraps them in its usual data structure by obtaining the necessary meta-data from their declaration. Thus, contrary to intuition, raw arrays are among the more expensive argument types in Babel. The second case are arrays with column or row order specifications (`ordered array`). Babel does whatever necessary to comply with these specifications, which might result in a deep copy of the actual data. Note, that these cases do not benefit from any of the optimizations presented in this paper. In many cases, the potential improvement would be rather small compared to the costs for copying the array. Thus, our current implementation disregards these cases and unconditionally falls back to the default client stubs. The user should be aware of the potential costs when using these features.

Considering C++, the costs for language interoperability are more significant. Even for simple data types, the costs compared to a native call increase by a factor of about 5x. For arguments requiring additional conversion, this only gets worse, *e. g.*, passing an interface is about 45 times slower than the native equivalent; for strings, the overhead is – depending on its length – more than two orders of magnitude.

Our system leaves the trade-off for the various caching strategies to the user's discretion; see Table II for an overview. Depending on the characteristics of the applications, any of those techniques may be preferable. The user

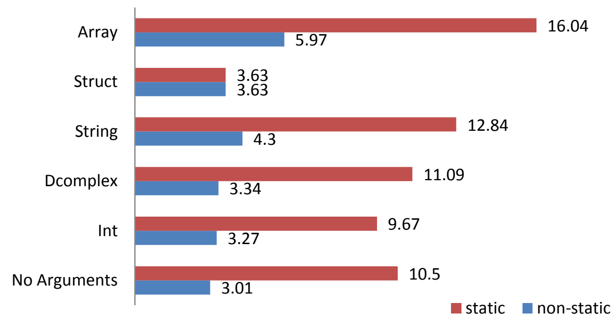

Figure 5. Speedup for Python method invocations compared to previous versions of Babel.

can select among them by setting pre-processor flags at compile time.

The static memory overhead per method is relatively small and usually of no concern to the user. The dynamic memory overhead per object and method for the implementation of fast delegates may be more of an issue for very lightweight objects. The user may choose to spend no memory at all ("no caching"). Even in this case, applying our techniques is always a clear win compared to the costs of a generic Babel call if both caller and callee are implemented in C++.

If the user is willing to spend some memory, we can do considerably better. There is almost no regression for non-native calls and native Babel calls cost about the same as native virtual C++ method invocations. For `gcc`, we can take advantage of a non-standard C++ extension and reduce the costs to that of a simple function pointer invocation. Otherwise, we have to pay the costs for a less efficient member function pointer invocation. The memory overhead is two machine words per method in the first case, and the costs of a regular pointer plus a member function pointer otherwise. In the case of eager caching, we pay the costs for the initialization of these fields at object creation time. For lazy caching, we do the same at the first invocation, requiring an additional dynamic branch. However, the differences are rather small as modern architectures successfully hide these costs in their branch prediction units.

In both cases, the speedup compared to regular native Babel calls ranges from about 5x for simple data types up to roughly 125x for strings. Thus, by spending a small amount of memory, we can effectively eliminate the runtime overhead for language interoperability completely for native method invocations without loosing flexibility.

Similar speedups could be achieved for Python; see Figure 5. However, while static and non-static methods behave similar in C++, there is a rather large difference for our Python implementation. The reason is that we do the dynamic lookup at every invocation for regular Python objects while we do it only once at object creation type for static methods. The same technique can also be applied to non-static methods, but would involve allocation and initialization of a local dictionary per object (as for regular Python objects).

V. Outlook and Future Work

The concepts proposed in this article nicely translate to other pairs of languages, but require some language-specific effort. C++ and Python where the most tempting targets to start with, but we plan to add support for further language pairs in the near future. In general, our approach is effective for most languages supporting some kind of dynamic dispatch, including newer revisions of Fortran (2003 or later).

Apart from these fine grained optimizations, we are pursuing ideas that require more fundamental changes to Babel by allowing for a more flexible representation of the IOR. These changes would allow to optimize the generated glue code for the particular subset of languages actually involved in an application.

References

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

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

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

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

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

[6] 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.

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

[8] 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.

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

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

[11] M. P. I. Forum, "MPI: A message-passing interface standard version 2.2," http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, 2009.

[12] O. A. R. Board, "OpenMP application program interface version 3.0," http://www.openmp.org/mp-documents/spec30.pdf.

[13] D. R. Butenhof, *Programming with POSIX(R) Threads*, 1st ed. Reading, Mass.: Addison-Wesley, 1997.

[14] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "A nonuniform memory access programming model for high-performance computers," *Journal of Supercomputing*, vol. 10, no. 2, pp. 197–220, 1996.

[15] "The cooperative parallelism project," https://computation.llnl.gov/casc/coopParallelism/.

[16] D. Jefferson, "Relationship between Co-op and MPI-2," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-TR-225783, November 2006, https://computation.llnl.gov/casc/coopParallelism/IM_340475converted.pdf.

[17] J. Knap, N. R. Barton, R. D. Hornung, A. Arsenlis, R. Becker, and D. R. Jefferson, "Adaptive sampling in hierarchical simulation," *International Journal for Numerical Methods in Engineering*, vol. 76, no. 4, pp. 572–600, 2008. [Online]. Available: http://dx.doi.org/10.1002/nme.2339

[18] N. R. Barton, J. Knap, A. Arsenlis, R. Becker, R. D. Hornung, and D. R. Jefferson, "Embedded polycrystal plasticity and adaptive sampling," *International Journal of Plasticity*, vol. 24, no. 2, pp. 242–266, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.ijplas.2007.03.004

[19] J. V. Bernier, N. R. Barton, and J. Knap, "Polycrystal plasticity based predictions of strain localization in metal forming," *Journal of Engineering Materials and Technology*, vol. 130, no. 2, p. 021020, 2008. [Online]. Available: http://dx.doi.org/10.1115/1.2884331

[20] N. R. Barton, N. W. Winter, and J. E. Reaugh, "Defect evolution and pore collapse in crystalline energetic materials," *Modelling and Simulation in Materials Science and Engineering*, vol. 17, p. 035003, 2009. [Online]. Available: http://dx.doi.org/10.1088/0965-0393/17/3/035003

[21] "Scalable linear solvers project," http://computation.llnl.gov/casc/linear_solvers/.

[22] "COMPASS SciDAC-2 project," http://compass.fnal.gov.

[23] "Community surface dynamics modeling system," http://csdms.colorado.edu.

[24] "Welcome to the FACETS project," https://ice.txcorp.com/trac/facets.

[25] C. L. Janssen, I. B. Nielsen, M. L. Leininger, E. F. Valeev, J. P. Kenny, and E. T. Seidel, "The massively parallel quantum chemistry program (MPQC), Version 3," Sandia National Laboratories, Livermore, CA, USA, 2008, http://www.mpqc.org.

[26] J. P. Kenny, C. L. Janssen, E. F. Valeev, and T. L. Windus, "Components for integral evaluation in quantum chemistry," *Journal of Computational Chemistry*, vol. 29, no. 4, pp. 562–577, 2007. [Online]. Available: http://dx.doi.org/10.1002/jcc.20815

[27] C. L. Janssen and I. M. B. Nielsen, *Parallel Computing in Quantum Chemistry*. Boca Raton, FL: CRC Press, April 2008.

[28] S. R. Kohn, G. Kumfert, J. F. Painter, and C. J. Ribbens, "Divorcing language dependencies from a scientific software library," in *PPSC*, 2001.

[29] L. C. Mcinnes, B. A. Allan, R. Armstrong, J. Steven, D. E. Bernholdt, T. L. Dahlgren, L. F. Diachin, M. Krishnan, J. A. Kohl, J. W. Larson, S. Lefantzi, B. Norris, S. G. Parker, J. Ray, and S. Zhou, "Parallel PDE-based simulations using the common component architecture," 2006.

[30] O. Dahl and B. Myhrhaug, "Simula implementation guide," Norwegian Computing Center, Oslo, Norway, Tech. Rep. Publication S47, Mar. 1973.

[31] M. Ellis and B. Stroustrop, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[32] British Standards Institute, *The C++ Standard: incorporating Technical Corrigendum 1: BS ISO*, 2nd ed. Wiley, 2003.

[33] D. Clugston, "Member function pointers and the fastest possible C++ delegates," Online Article, 2004, http://www.codeproject.com/KB/cpp/FastDelegate.aspx.