Open C++ and Its Optimization (Extended Abstract)

Shigeru Chiba* Takashi Masuda
Department of Information Science, University of Tokyo
E-mail: {chiba,masuda}@is.s.u-tokyo.ac.jp
(*OOPSLA'93 Workshop on Reflection*)

1 Introduction

Open C++ [1] is a reflective C++ that allows us to implement new language primitives on top of the language. Those new primitives are implemented without rebuilding a compiler; meta code written in C++ implements them. By implementing new primitives, programmers can extend Open C++ to be a distributed language and a parallel language as they like.

To implement a new language primitive, a metaobject is created and it controls the behavior of an object. Different metaobject classes implements different kinds of language primitives. By choosing a metaobject class, an object is made to be a distributed shared object and a proxy of a remote service, for example. Programmers can define their own metaobject class easily by using inheritance to obtain the most suitable primitive for their applications. Using the meta-level system of Open C++, programmers can build a class library that has functionality impossible within the confines of plain C++.

Although an Open C++ program runs fast enough to implement distributed applications, its performance is about 10 times slower than a C++ program's, in terms of reflective code. The current Open C++ compiler has an optimization option, which improves the performance of Open C++ to be near to that of C++ in the best case. The optimization technique that the Open C++ compiler uses is based on partial evaluation. It collapses base and meta levels to be flat.

*JSPS (Japan Society for the Promotion of Science) Fellow-DC

2 Open C++

Open C++ exposes the implementation of method calls and variable access to an object. Programmers can alter it by a metaobject to obtain a new language primitive. Since a lot of language primitives provided by existing distributed langauges are extensions of method calls and variable access, the authors believe that the extensibility of Open C++ covers various kinds of language primitive for distributed computing. In Open C++, other mechanisms such as an inheritance rule is not changeable. To achieve good performance, Open C++ omits the ability to change mechanisms irrelevant to distributed computing.

Open C++ has a simple MOP (metaobject protocol) to alter the implementation of method calls and variable access. Programmers define a metaobject according to this MOP. Open C++ MOP is simple and easy to understand; non-experts of Open C++ can use it easily. A metaobject basically traps method calls and variable access to its object, and carries them out in stead of the default implementation embedded in a C++ compiler.

At meta level, a metaobject handles the actual argument list and return value of a method call, and a value assigned to and read from a variable. It handles them through a C++ object independent of the types of the method and variable. This C++ object provides an abstract interface for programmers to manipulate non-first-class entities at base level. In Open C++, actual argument lists, return values, and the values of variables are reified to be first-class entities at meta level. They can be transferred to other hosts through a network to implement remote procedure calls, for example. Programmers can define a metaobject independently of the class of the object that the metaobject controls. This feature of Open C++ is useful to increase the reusability of meta code. It also enables clear separation of distributed notions from base-level code.

Metaobject classes constitute a class library. Each language primitive is implemented as a subclass of class `MetaObj`, which is a root of all classes. A metaobject implementing a distributed shared object, for example, controls variable access to its object to keep the object holding a replication of shared data. Another metaobject may implement remote procedure calls. It controls its object so that the object acts as a proxy of a remote service, or as a client stub of remote procedure calls.

The difference between meta code of Open C++ and library programs of plain C++ is that meta code treats internal data, such as argument lists, as first-class entities. Library programs of C++ cannot directly treat such internal data without programmer's assistance. For example, suppose a case that a programmer uses a C++ library that would implement remote procedure calls. The programmer must describe code by hand to convert an actual argument list to a network

message for each procedure. Also, the programmer must describe code to dispatch a received message to an appropriate procedure at the server side. These defects are because C++ cannot treat an argument list (not an element of the list) and procedure names as first-class entities. Open C++ can treat directly them at meta level, however. Programmers can describe meta code that specifies how to make a network message and send it, and how to dispatch a received message, but that is commonly used for every method. A programmer who reuses that meta code does not have to describe additional code for each method.

3 Discussion on Overhead

According to our experiment, a method call implemented by a metaobject is 5 or 8 times slower than that of plain C++, . Variable access is 35 times slower. These overheads are negligible compared with the latency time of a network such as Ethernet. The network latency time is between several hundreds microseconds and some milliseconds, whereas a null method call of plain C++ takes only a microsecond.

To minimize the overhead due to reflection, furthermore, Open C++ has programmers specify whether or not to use reflection for every object, method, and variable. If reflection is not used, then the object, method, and variable run as fast as plain C++'s ones. Since well-designed applications need reflection only for a small number of objects, which would act as gateways from other hosts, distributed applications developed in Open C++ run as efficiently as applications developed in plain C++.

Although the overhead involved by Open C++ is negligible in terms of distributed computing, it is not true if Open C++ is used to describe, for example, parallel computation on a multicomputer. A multicomputer has a fast network connecting processor nodes, and its latency time is a few hundreds microseconds or less. The rest of this paper discusses a technique to reduce the overhead due to reflection much more.

4 Optimization

The overhead of Open C++ mainly results from reifying and reflecting. They are operations that convert internal data to first-class entities available to a programmer, and cause side-effects at base level by using the converted data. Since those operations are principle ones in reflection techniques, an optimization technique for those operations will be applicable to other reflective languages.

The reify operation of Open C++ is to convert internal data, such as the argument list and

return value of a method call, and values assigned to and read from a variable, to first-class entities. At meta level, these reified data may be transferred to other hosts, and also used to cause side-effects at base level, for example, they are used to execute a method call and variable access at base level. Causing side-effects at base level is done by the reflect operation of Open C++.

The basic scheme of the optimization technique of Open C++ is the use of partial evaluation to collapse base and meta levels. By collapsing multiple levels, internal data that was converted to first-class entities are directly manipulated as it is, by collapsed meta code. Actual code corresponding to reifying and reflecting is eliminated so that the overhead is reduced considerably.

If optimization is specified, the Open C++ compiler does partial evaluation and produces specialized code for each method and variable controlled by a metaobject. Recall that meta code is described independently of each method and variable at base level. This generality is necessary in terms of code reusability, although it causes the overhead if optimization is not applied.

The specialized code is produced using the following procedures:

- specializing a pair of reify and reflect:
  To collapse base and meta levels. If a reify operation converts internal data and a reflect operation causes side-effects with that converted data without modification, then that pair of reify and reflect is specialized in terms of the processed internal data. Because, as a result of the specialization, the reflect operation knows which internal data is processed, it can directly use the internal data to cause side-effects at base level. Since the reflect operation does not need to re-convert the reified internal data, its performance is improved. For example, if a pair of reify and reflect processes the argument list of a method call, then the reflect operation executes a method call directly using the argument list on a stack frame and registers.

- deferring reify:
  To defer a reify operation until the reified data is necessary. If the reified data is not used, then the reify operation is eliminated. Internal data are reified when they are transferred to other hosts through a network, for example. In such a case, a reify operation is not removed because internal data cannot be translated to a network message without reifying.

- eliminating unnecessary branches:
  To eliminate branches that are unnecessary as a result of specialization. Meta code often branches depending on which method and variable it controls. This procedure eliminates those branches if possible. This is helpful to analyze flow of meta code so that the other procedures are applicable much more.

Although applying partial evaluation is not simple work, the simplicity of Open C++ MOP enables partial evaluation by those simple procedures. According to our preliminary experiment, the performance of a method call is improved to be mostly the same as that of a plain C++ method call. The performance of variable access is also improved to be 10 times slower than that of plain C++ one. Recall that a method call is 10 times slower and variable access is 35 times slower, unless the optimization is not done.

Some reify and reflect operations leave after partial evaluation. This is because meta code often need reified data, which the meta code does not reflect. The overhead due to those operations results from not reflection but a language primitive itself implemented on top of Open C++. It appears even if that language primitive is implemented without reflection.

Specialization of a pair of reify and reflect is not applied if reified data are modified by meta code. In most of Open C++ programs, however, meta code deals with reified data as it is, without modification. This is because meta code is described independently of methods and variables at base level. Meta code seldom modifies reified data. This strongly depends on the actual type of the reified data.

5 Conclusion

This paper showed a reflective C++, named Open C++. This language is extensible to support various language primitives for distributed and concurrent computing. Although the overhead due to a reflective facility of Open C++ is negligible compared with the latency time of networks like Ethernet, this paper showed an optimization technique that reduces the overhead much more, so that Open C++ can support applications that run on a multiprocessor machine with a faster network.

The first version of Open C++ has been implemented, and it can be obtained by anonymous ftp from `utsun.s.u-tokyo.ac.jp` (133.11.11.11). The products includes a metaobject class library that implements typical language primitives, such as remote procedure calls and distributed shared objects, on top of Open C++.

References

[1] Chiba, S. and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *Proc. of the 7th European Conference on Object-Oriented Programming* (to appear), 1993.