Compiling Object-Oriented Programs for Distributed Execution

Prakash K. Muthukrishnan
Barrett R. Bryant
Computer and Information Sciences Dept.
University of Alabama at Birmingham
Birmingham, AL 35294-1170, U.S.A.
{prakash, bryant}@cis.uab.edu

Abstract

Techniques based on formal semantics are discussed for automatically generating parallelizing compilers for object-oriented programming languages. The denotational semantics of object-oriented programming languages are used to derive the control and data dependencies that exist within programs and this information may then be used to produce parallel object code by a compiler. This approach also easily facilitates proving the correctness of transformations performed on the source programs. Furthermore, since compilers may be automatically generated from denotational semantics specifications, the implementation of these parallelization techniques will be automatic and hence language independent.

1. Introduction

Object-Oriented (OO) programming is a paradigm that provides linguistic support for objects, classes and inheritance [14]. It supports the software engineering principles of data abstraction, information hiding, modular design, and code reuse, and it allows the modeling of real world objects in a natural fashion. Since objects are independent entities that communicate through message passing, the object-oriented paradigm lends itself naturally to the development of distributed systems [16]. This has been pursued primarily by developing object-oriented languages with explicit concurrency constructs called concurrent object-oriented programming languages. For surveys of general principles, the reader is referred to [13]. The development of concurrent object-oriented programming systems have been complemented by efforts in defining the formal semantics of such systems [1, 12, 16]. There has been only very little work done in automatic parallelization of OO languages [3, 15].

We use formal semantics specification in deriving the parallelism that is available in object-oriented programs, which are inherently parallel, and propose a tool which a software developer can use to develop parallel applications by just writing sequential object-oriented code. Such technology has not been applied for Object-Oriented Programming Languages (OOPs). We propose a methodology that extends existing techniques in automatic compiler generation [8] and parallelizing compilers [17]. One immediate benefit of such a tool is that it could be used to enable an existing object-oriented programming language, such as Smalltalk [4] or C++ [11], to be directly compiled into parallel code. The result would be that applications programmers would not have to learn a new object-oriented language in order to implement a distributed object-oriented system.

The paper is structured as follows. In Section 2 we present SmallC++, a subset of C++ that we believe to be tractable for developing a prototype of a parallelizing compiler. In Section 3 we present the semantics of SmallC++. It is defined by a two phase translation that corresponds to the generation of the parallel target code. In the first phase SmallC++ is translated into combinator code. In the second phase, semantics preserving transformations are performed on the combinator representation that replace all potential parallelism by parallel combinators and communication primitives. We use the Tuple Space model of concurrency [2] as the underlying model for object interaction. The result of the transformations is an executable parallel program with well understood semantics. And finally in Section 4 we present the summary of our project and future work.

2. SmallC++

We have developed a language called SmallC++ [9] in order to test our proposed method. SmallC++ is a distilled version of C++ with only the core object-oriented language features. A program in SmallC++ contains a set of class declarations and a main function. A class declaration contains definitions of data members and member functions and
is fully encapsulated. Class hierarchies, which are used to relate various classes, can be declared using inheritance. Apart from methods declared in classes, regular C language like functions can be declared at the global scope level.

The set of commands that can be used in SmallC++ include assignment, if, for and compound statements. Also input and output are available in the form of cin and cout stream operators. Expressions in SmallC++ support the regular arithmetic operators, and member selection using the dot (.) and a combination of the this pointer and the arrow operator (→).

The following is an example program written using SmallC++.

```cpp
class Point {
    private:
        int x;
        int y;
    public:
        Point(int a, int b) { x = a; y = b; }
        intGetX () { return x; }
        intGetY () { return y; }
};

int main () {
    Point obj1 (3,4);
    Point obj2 (5,6);
    float d;
    d=sqrt(square(obj1.GetX()- obj2.GetX())+
            square(obj1.GetY()- obj2.GetY()));
    // Note all four method invocations
    // can execute in parallel
}
```

### 3. Semantics Based Parallelizing Compilation

In this section we outline the process of generating distributed target code for SmallC++ using semantics based compilation techniques. As this exactly parallels our definition of the semantics of SmallC++, the description of code generation is interwoven with the definition of the semantics.

In semantics-based compilation, type checking and code generation phases of the compiler are driven by the formal semantics of the language. Typically, the formal semantics are given in terms of a combinator language, where combinators are operators with direct representations in conventional target machine languages (e.g. arithmetic and logical operations, conditional and iterative constructions, etc.). Sequential code generation thus corresponds to the formal definition of the semantics of the language. Type checking can be performed either by evaluating the combinator code or by using any standard type checking algorithm.

In order to generate distributed code, we define a set of parallel combinators corresponding to the underlying distributed architecture.

We have chosen denotational semantics [5] as our formal semantics basis. The major issues involved in defining SmallC++ denotationally are: 1) the need for each object to have its own independent denotation with semantics for communications with other objects through message passing; 2) for methods within the same object to be able to share common instance variables stored within the object; and 3) to model inheritance in the presence of dynamic binding. The denotational semantics of these constructions must allow all possible parallelism while constraining the parallelism through the synchronizations required by the object semantics.

We define a novel two step process for giving the denotational semantics of SmallC++. First SmallC++ is translated into a sequential combinator language. This provides a sequential interpretation of SmallC++, with a well understood denotational semantics. Next, we perform semantics preserving parallelizing transformations on this code. Parallelism is encoded using a set of parallel combinators with communication primitives. The result of this definition is a parallel program that can run in a distributed tuple space model with communication represented by Tuple Space operators. Because the process is semantics based, this program is provably semantically equivalent to the sequential OO program.

#### 3.1. Sequential Semantics

Our denotational language is a set of combinators with a combinator for every major semantic action in SmallC++, such as looping, sequential execution, conditional execution, memory access, and type checking. Figure 1 summarizes the set of sequential combinators we designed to implement SmallC++.

Of these combinators, message-send is most directly related to the semantics of objects. All other aspects of objects will be parameters to other combinators. An object is encoded as a store-like function that returns the denotations of instance variables and methods for that object. Classes define templates for constructing these objects and are denoted by the appropriate constructor functions.

Figure 2 is the denotation of the SmallC++ program introduced in Section 2 after compilation. It has various sequential combinators like create-object and message-send. For brevity, we do not change the identifier names into their actual denotations.
access-array, update-array
access-field, update-field
access-member, update-member
create-object
copy-block
assign
compose
if
for
call
message-send
return
fix
ref
deref
plus, minus, times,
slash, equal, not-equal,
less, less-equal, greater,
greater-equal, and, or, not

access and update array elements
access and update components of structures
access and update instance variables of objects
create an object
copy structures and arrays
define an assignment operation
sequential composition
conditional (must have “then” and “else” parts)
for-loop (must have an index, initial value, termination condition
and loop body)
procedure call
send a message to an object (similar to “call”)
return a value
the fix-point combinator, represents recursively defined functions
call-by-reference parameter
access a call-by-value parameter
standard arithmetic, relational, and logical operators

Figure 1. Sequential Combinators

3.2. Parallel Semantics

The second step in defining the semantics is applying
a set of semantics preserving parallelizing transformations
that encode SmallC++ by a parallel combinator language
augmented with communication primitives. This encoding
is executable on a distributed platform. The transformation
is performed using flow analysis, dependence analysis, and
parallel code generation. We describe the parallel combinat-
ors, the Tuple Space model of communication, and the pro-
posed transformations below.

3.3. Parallel Combinators

Parallelism is encoded using parallel combinators. Par-
allel combinators were introduced by [6] to define the gran-
ularity of parallelism, and [7] to generate parallel functional
code. Our approach is most like [7] in that the combinators
themselves represent the basic instruction set of the paral-
lel machine model being used, in our case the Tuple Space
model. Some examples of parallel combinators are given in
Figure 3. The set of parallel combinators is still under de-
velopment. So, only a high-level set is illustrated here.

It is up to the implementation for a particular target ma-
chine to determine how these combinators are to be inter-
preted. For example, on a sequential machine, they would
have no effect; in a Tuple Space system, they would be tran-
slated to the process control instructions of that system.

3.4. Tuple Space

We use the generative communication model of distrib-
uted computing or Tuple Space (TS) for the implementa-
tion of interprocess communication. TS represents commu-
nication among distributed processes by tuples. Formally,
a tuple is an n + 1-tuple in which the first element of the
tuple is a name, and the remaining n elements are param-
eters, either formal or actual. The actual parameters repres-
ent data to be sent and the formal parameters represent place
holders. The model is dynamic, with the tuple space of a
program changing as the program executes. For example,
if A and B are objects, A sending a message to B will gener-
ate a new tuple in the space. When B is ready to receive this
message it removes the tuple and executes. These actions
are encoded by the functions out, which puts a new tuple
in the space; in, which removes a tuple from the space; and
read, a non-destructive in.

TS provides a good interprocess communication model
for an OO environment for several reasons. First, tuples,
like objects, can have both data members and functions.
Thus they provide a natural representation of objects.
Second, a fault-tolerant Tuple Space implementation is
available [10]. Third, a tuple can consist of a name and a
function, facilitating the creation of a process which can ex-
cute in parallel. We call such tuples message tuples and
they can be encoded using the eval operator that was in-
troduced in [10]. Message tuples can return a value in the
TS, that can be used by other processes, at the end of its ex-
ecution. Thus, a method and data member can be treated
(compose
  (create-object Point obj1 3 4)
  (create-object Point obj2 5 6)
  (create-object float d)
  (assign d
    (call sqrt
      (plus
        (call square(minus (message-send obj1 GetX)
          (message-send obj2 GetX)
        ))
        (call square(minus (message-send obj1 GetY)
          (message-send obj2 GetY)
        ))))))

Figure 2. Denotation of the example program in sequential combinators

| parallel   | argument expressions can be evaluated in parallel |
| sequence   | argument expressions must be evaluated sequentially |
| distribute | argument objects can be distributed |
| cluster    | argument objects should not be distributed |
| for-all    | parallel for-loop (must have an index, initial value, termination condition and loop body) |

Figure 3. Parallel Combinators

equally in our implementation. Finally, using `eval`, a process may request the execution of a program (a method, for example) on a specific node or allow the system to select the node within the TS network which has the least number of active processes thus providing an automatic load balancing mechanism.

3.5. Generating the Parallel Semantics and Code

The basic structure of a parallelizing compiler for an object-oriented programming language is similar to that for more conventional languages [17]. However, there are some major differences in the nature of the components, caused by object-orientation. Apart from the regular loop-parallelism that is available in Fortran-like languages, there are at least three other forms of concurrency possible within an object-oriented system. Inter-object concurrency refers to different objects carrying out different activities at the same time. Intra-object concurrency refers to a single object executing several methods simultaneously. Thirdly each of these methods could themselves be carrying out several operations in parallel. These forms of concurrency can lead to the following sorts of placement on a distributed architecture.

- Objects that are independent of one another can be placed on separate processors and can execute concurrently.
- Methods in the same object that are independent can also be located on separate processors so that they can execute concurrently.
- Methods dominated by statement level parallelism can be implemented on a multiprocessor designed to exploit fine grained parallelism.

Thus, an object may have several threads of control, each corresponding to different types of concurrent execution.

The primary components of the parallelization of OO programs are flow analysis, dependence analysis, and parallel code generation. A general overview is presented in Figure 4.

Flow analysis creates a graphical representation of the flow of data among components of a program. Unlike the flow analysis performed on non-OO programs where nodes represent simple statements, the flow graph we create has three types of nodes: (1) individual methods, (2) a collection of methods within a class or object, and (3) all the methods of an OO program. The primary complexity introduced by the OO paradigm is determining the precise instance variables and methods being referenced in the presence of polymorphism and inheritance.

Dependence analysis uses the flow graph to determine the data dependencies and the interconnectedness among the objects in the program. Because we are working with an OO programming paradigm, we must consider both the dependencies within methods and the dependencies that result from message passing. The first type is handled by constructing a standard dependence graph for the code within each method. We determine the interconnections and flow of data among objects using a second type of dependence graph we call a message flow graph.

Parallel code generation uses the flow and dependence information to partition the objects into processes in a manner that reduces communication costs. This includes determining whether an object process should be executed synchronously or asynchronously. Furthermore, we may generate code that automatically controls communication between processes by increasing the amount of data transferred in
Figure 4. Overview of Denotational Semantics-Directed Parallelization

The following is the Tuple Space realization of the code shown in Figure 5.

```
out ("Pointobj1", 3, 4);
out ("Pointobj2", 5, 6);
float d;
eval ("dtmp1", tmp1 = GetX (Pointobj1),
    tmp2 = GetX (Pointobj2),
    tmp3 = GetY (Pointobj1),
    tmp4 = GetY (Pointobj2));
eval("dtmp2", tmp5 = square(tmp1 - tmp2),
    tmp6 = square(tmp3 - tmp4));
```

Figure 5. Denotation of the example program in parallel combinators

The sequential combinator code produced for the example program in Section 3.1 is now transformed to the program shown in Figure 5 after performing flow and data dependence analysis. This program contains the parallel combinators identifying parts of the code that can be executed in parallel. The creation of the two point objects (and the distance variable) are done in parallel. Various portions of the calculation of the distance between the two points are also done in parallel.

The following is the Tuple Space realization of the code shown in Figure 5.

```
out ("Pointobj1", 3, 4);
out ("Pointobj2", 5, 6);
float d;
eval ("dtmp1", tmp1 = GetX (Pointobj1),
    tmp2 = GetX (Pointobj2),
    tmp3 = GetY (Pointobj1),
    tmp4 = GetY (Pointobj2));
eval ("dtmp2", tmp5 = square(tmp1 - tmp2),
    tmp6 = square(tmp3 - tmp4));
```
tmp6 = square(tmp3 - tmp4));
eval("d",d = sqrt (tmp5 + tmp6));

The Tuple Space representation outs two tuples, one for each Point object. The names "Pointobj1" and "Pointobj2" are keys that are used to identify the tuples, created by concatenating the Class Name and the Object Name (we would wish to use a more unique key creation algorithm in practice). The first eval forks four processes each executing one method and returning a value in the temporary variables. The second eval forks two more processes, each using the previously computed temporary values. The final eval forks a process which computes the distance and returns it in d. In essence, the computation in the above program is done in just three steps - one step for each eval, ignoring the outs in the first two statements.

4. Summary

The implementation of our parallelizing compiler exists at present in only a prototype form. We have already defined a denotational semantics of SmallC++. A compiler translating SmallC++ programs into denotational code based on our set of combinators has been implemented. Currently we are working on representing the denotational code in the form of dependence graphs. Our next objective is to study the graphs and complete the set of parallel combinators. In the final stage we plan to compile the SmallC++ programs to Fault Tolerant Tuple Space that was implemented by [10] on a network of Sun workstations. This implementation can easily be extended to a variety of heterogeneous platforms. This project facilitates the translation of a sequential program into an equivalent, semantically provable parallel program that can be executed in a parallel fashion.

References


