Increasing Cross-Domain Call Batching using Promises and Batched Control Structures

Quinton Y. Zondervan

June 1995

This work was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136.

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Increasing Cross-Domain Call Batching using Promises and Batched Control Structures

Quinton Y. Zondervan

Abstract

In a client-server system, it may be possible for the client to corrupt server data through unsafe access methods or programming errors. A common method for protecting the server data is to separate the client and server into distinct protection domains, e.g. Unix processes. Communication between client and server is restricted to a well-defined set of cross-domain calls and results are passed as opaque pointers to server data. Unfortunately, cross-domain calls are expensive and seriously affect the performance of the system. In his work on Batched Futures, Bogle presents a mechanism for batching cross-domain calls, thus amortizing the overhead of the domain crossing across the batch. Bogle showed that the improvement in performance is directly proportional to the average size of a batch (batching factor). This thesis describes two mechanisms that can be used to improve the batching factor: basic value promises and batched control structures. Basic value promises allow the batching of calls that return basic values (e.g. integer, boolean, character and real). Batched control structures allow the batching of simple control flow statements such as if statements, while loops and for loops. Both mechanisms have been implemented in Thor, an object-oriented database system. This thesis presents performance results showing the effectiveness of both mechanisms in increasing the performance of cross-domain calls.

Keywords: futures, promises, control structures, batching, protection domains, cross-domain calls, object-oriented databases.

Acknowledgements

My gratitude and appreciation to the following people:

Prof. Barbara Liskov for supervising this thesis in particular and aiding me in my graduate career in general.

Howard Trickey, my AT&T CRFP advisor.

The past and present members of the Programming Methodology Group for their technical support and friendship. Special thanks to: Philip Bogle, Atul Adya, Mark Day, Andrew Meyers, Umesh Maheshwari, Sanjay Ghemawat, Dorothy Curtis and Miguel Castro.

Charles Isbell for being a great friend and invaluable resource.

Kelsey Jordahl for being a reliable friend and roommate.

My wife, Radhika Nagpal, for being a wonderful companion. Without her love and support this would have been a much harder task.

My parents, though far away, still a never ending source of support and inspiration.

Contents

List of Figures

Chapter 1

Introduction

In a client-server system, data integrity of the server can be guaranteed by separating the client and the server into distinct protection domains. Communication occurs through a well-defined interface that ensures that the client cannot corrupt the data stored at the server or otherwise interfere with the server's operation. However, crossing a domain boundary can be expensive. If the client and server are distinct processes running on the same machine, each call requires a context switch, which is much slower than a direct function call [1]. When the client and server are running on separate machines, and need to communicate across a network, the cost of a context switch is even worse.

In previous work, Bogle [3] presented a mechanism, called **batched futures**, for sending calls to the server in batches. Normally, each call is sent to the server and evaluated. Results are either **handles** which are references to object stored at the server, or basic values such as integers or booleans. The batched futures mechanism allows calls that return handles to be batched. Instead of sending such a call individually, and waiting for the result, the client places the call in a batch, and generates a place holder for the result called a **future**. The future can be used as an argument to later calls. Batching a set of calls improves the performance of cross-domain calls by amortizing the cost of the domain crossing over the number of calls in the batch. Thus, the performance improvement is directly related to the number of calls in the batch.

The batch size for Bogle's mechanism is limited, however, for two reasons. The first is that it allows only calls that return handles to be batched. Calls that return basic values are always executed immediately: at that point any batched calls are run followed by the call returning the basic value. Since calls returning basic values occur frequently in client code, batch sizes are necessarily rather small.

The second reason is that only straight-line code can be batched. Bogle's mechanism allows a sequence of client calls to be batched, provided they return handles. However, the presence of a control structure whose condition depends on the result of a call necessarily interrupts the batch. This is because the call must be made immediately so that the client code can use its result to determine the control flow of the program. Thus, for example, a loop controlled by a test that examines the state of a server object cannot be run as a single batch. Instead, the batch will be terminated on each iteration of the loop.

In his thesis, Bogle suggested some ways of overcomming some of these limitations. This work extends the batching mechanism along the lines suggested by Bogle. First it provides a mechanism that allows calls that return basic values to be batched by generating a special kind of future for the result, called a **promise**. A **claim** operation is defined on promises, which allows the client to extract the actual value of the promise at some later time. Promises allow the client to batch calls that return basic values, but whose results are not of immediate interest, thus increasing the size of the batch and improving the overall performance.

Second, the thesis provides a way of batching control structures by means of a new mechanism, called **Batched Control Structures**. Batching a control structure does not interrupt the current batch, thus allowing calls preceding and following the control structure to be placed in the current batch as well.

When a batched control structure such as a while loop is processed by the server, it is parsed once and executed multiple times. While parsing the loop, the server generates a highly efficient representation for the loop, which is then evaluated. Evaluating this representation is more efficient than processing individual calls sent by the client. Thus, batched control structures improve performance in two ways: by increasing the batch size, and by preprocessing the batch before running it, thus reducing the cost of actually running the calls in the batch.

The batched control structure mechanism supports two kinds of looping structures: while loops, and foreach loops. The latter kind of loop makes use of a server iterator. An iterator is a special kind of routine that produces multiple results; it yields the results one-at-a-time and is used in a for loop that does an iteration for each yielded value [11]. Prior to the work in this thesis, there was no way for client code to make use of server iterators in our system; the foreach mechanism overcomes this limitation.

1.1 Roadmap

- Chapter 2: Describes Thor, an object oriented database system and the context for this work.
- Chapter 3: Describes batched futures, the background for most of this work.
- Chapter 4: Describes the design and implementation of Basic Value Promises in Thor.
- Chapter 5: Describes the main contribution of this thesis, Batched Control Structures.
- Chapter 6: Describes the implementation of Batched Control Structures.
- Chapter 7: Presents performance results for Batched Control Structures
- Chapter 8: Describes some possible extensions to the work presented in this thesis, and gives some conclusions.

Chapter 2

Thor Client Interface

The context for this work is Thor [12], a new object-oriented database system that provides persistent and highly available storage for its objects. The first half of this chapter describes some of the relevant features of Thor and the basic system architecture. The rest of the chapter describes the mechanism by which client programs interact with Thor.

2.1 Thor concepts

To the client, Thor provides a universe of **objects**. These objects are fully encapsulated, so that their internal representation is not accessible to the client. Clients can obtain **references** to Thor objects, which can be used to pass the objects as arguments to **method** calls or calls to **routines**. In addition, **basic value** arguments can also be sent. In Thor, basic values are integers, booleans, characters and real numbers.

Methods and Routines A method is a procedure that is defined as part of a **type**. A method has an implicit first argument which is an object of the type for which it is defined. A method is invoked on an object of its type. A routine is a stand-alone procedure, not associated with any type.

Iterators In addition to procedures, Thor supports **iterators**, introduced in the CLU programming language[11]. Rather than returning a single value, an iterator **yields** a new value each time it is called. Iterators are called from within a special **for** construct. Both routines and methods can be iterators.

Object references A reference to a Thor object is called a **handle**. Handles are returned to the client as a result of method calls performed by Thor on behalf of the client. These handles can be used as arguments to subsequent method calls. Handles are valid only for the duration of a particular session between the client and Thor.

Exceptions Thor routines and methods can return normal values, or signal an exception upon termination [11][9]. An exception indicates that the normal return values could not be computed.

Types The operations that can be performed on an object are determined by its type. Thor types are specified and implemented in **Theta** [8], a new object-oriented programming language being developed specifically for Thor.

A type consists of a set of method signatures. For example, here is the definition of the **DataBase** type in Theta, which will be used in later examples:

DataBase = **type**

fetch (user: string) **returns** (DataElem) **signals** (not_found)
% Find user in database, and return corresponding record.
% Signal not_found is user not in database.

new_user (user: string) **returns** (DataElem) **signals** (not_possible)
% Create new record for user, and return the new record.
% Signal not_possible if record cannot be created.

users () **yields** (DataElem)
% Yield the records for all users in database.

end DataBase

The keyword **signals** indicates the exceptions that can be signaled by the method. The keyword **yields** is used instead of **returns** in the case of an iterator.

Type hierarchy Theta allows subtyping. In Theta, a subtype S must have all the methods of its supertype T with compatible signatures [16, 2, 4], plus perhaps some additional methods. If S is a subtype of T, an object of type S can be supplied as an argument where an object of type T is expected.

2.2 Thor architecture

The Thor system is divided into the following components:

- A set of distributed storage servers that persistently store Thor objects. These are called **Object Repositories (OR)**
- A process called the **Frontend** that caches objects and executes calls to routines and methods on behalf of the client.

The client interface in Thor is defined by the interactions between the Frontend and the client program. If the client language is unsafe [1], the client and the Frontend must be executed in separate protection domains. In practice, that means that the client and Frontend

[1] A language is considered unsafe if it allows arbitrary pointer dereferencing as in C or C++

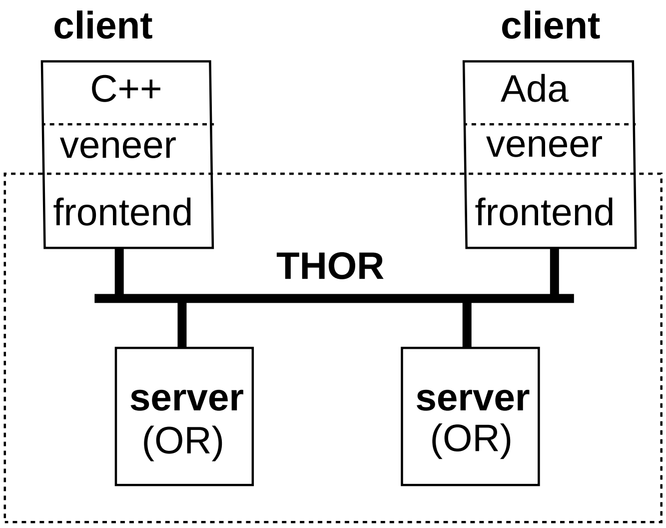

Figure 2-1: Thor architecture.

run as separate processes on the same machine, although it is also possible for the client and Frontend to run on separate machines.

Figure 2-1 illustrates the Thor architecture, with the client and Frontend running on the same machine. The client communicates with the Frontend using a language specific **veneer**. Veneers are described in section 2.4.

2.3 The Frontend

Each Frontend is dedicated to a single client application. Objects accessed by the client are fetched from the OR and cached at the Frontend [7].

The interface presented to the client by the Frontend includes the following commands:

- **lookup_wellknown** This command takes a string as its only arguments. The Frontend searches for the string in a table of "well known objects" and returns a handle to the object corresponding to the string argument.

- **invoke** This command is used to invoke a method or routine [2]. Arguments are the method name, and the arguments for the call, including the receiver.

The client makes calls to the Frontend by communicating either through a shared-memory buffer, or through a TCP/IP network connection. Generally, a client will begin its session by looking-up some "root" object in the database.

The Frontend continually waits for commands from the client. Probably the most frequently used command is the invoke command. The client sends the name of the method, and its

[2]The current implementation of Thor does not yet support routines, so we will not discuss them in the remainder of this thesis

arguments in the form of handles and/or basic values, to the Frontend. The handles are used to find the corresponding objects in the **handle table**, H, which is maintained by the Frontend. The name is used to find the method's signature, which is used to verify that the right number of arguments have been passed, and to type check those arguments. A call is type correct if the type of each argument is a subtype of the type listed for that argument position in the signature [3]. If the call type checks, the call is made, and the result object is mapped to a new handle in H, which is sent back to the client. Basic value results are not mapped to handles, but are sent back to the client directly. If the method resulted in an exception being signaled, the exception is send to the client. Handles are implemented as positive integers which index into the handle table H.

The logical interface at the Frontend is implemented by two different interfaces, a textual interface (ascii) and a binary interface. The ascii interface is suitable for direct user access (e.g. using the telnet protocol), or clients written in simple text processing languages. The binary interface provides higher performance and more direct encoding of values, and is used by C and C++ clients.

2.4 Veneers

One of the goals of Thor is to allow application programs written in different programming languages to share the same objects. This is accomplished by providing a **veneer** [3] for each programming language. The veneer extends the language with the necessary procedures to interact with Thor. Several veneers have been implemented for Thor, including C, C++, Emacs Lisp, Perl, TCL [3] and Modula3 [10]. In the remainder of this thesis we will limit our discussion to the C++ veneer, but the concepts explained here are not limited to the C++ veneer.

The veneer provides the client with an interface to Thor. The interface is implemented by a library of procedures. These procedures are generally stub procedures that call the corresponding remote procedures in the Frontend interface described in the previous section.

In addition, the C++ veneer maintains a mapping between handles and **stub objects**, which are C++ objects that mimic corresponding Thor objects. In the remainder of this section we will describe the stub object mechanism in more detail. For a more general discussion of veneers, see [3].

2.4.1 Veneer stubs

The veneer manages the set of handles currently referred to by the client. In the C++ veneer, these handles are encapsulated in **stub objects**, which mimic the Thor objects they represent. The veneer also provides a stub procedure for each Thor routine or method. In the C++ veneer, stub procedures are implemented as member functions of stub objects.

For each Theta type defined in Thor, a corresponding C++ class is generated automatically and included as part of the veneer. For example, here is the C++ class definition for the DataBase type mentioned earlier:

[3]As an optimization, basic value arguments are not type checked. This speeds up execution, and does not really violate type safety because Thor treats a basic value as an opaque 32 bit number, regardless of its type. Thus, any method defined on integers will execute when passed a real, bool or char, though an exception may be signaled in some cases.

```
class th_DataBase : public th_any
{
 public:
  th_DataBase() : th_any() {}

  th_DataBase(int handle_) : th_any(handle_) {}

  th_DataElem* fetch(th_string* user);
      /* signals(not_found) */

  th_DataElem* new_user(th_string* user);
      /* signals(not_possible) */

  th_DataElem* users();
};
```

Each stub object contains a handle to a Thor object. To invoke a method on the Thor object, the client invokes the method stub on the stub object. Thus, Thor method invocation looks like regular C++ invocation to the client programmer. The stub procedure marshals the arguments and sends the actual call to the Frontend. The Frontend executes the call and sends back the result. Results are either handles or basic values. The stub procedure unmarshals the result and returns it to the client. Result handles are encapsulated in new stub objects before they are returned.

The following code fragment illustrates how to call the `fetch` method on an object of the DataBase type:

```
th_DataBase *db = look_up_wellknown(``root'');
th_string *usr = new th_string(``qyz'');

th_DataElem *el = db->fetch(usr);
```

The first statement calls a stub procedure that calls the corresponding look-up procedure at the Frontend. In this case the root object is of the type DataBase. The resulting handle is wrapped in a stub object, and a pointer to that stub object is assigned to the variable `db`. The second statement creates a new Thor string with the value "qyz". A handle to the string object is returned by the Frontend, which is wrapped in a new stub object. The variable `usr` is assigned a pointer to this stub object. The last statement calls the stub function for the `fetch` method on the stub object `db` with argument `usr`, and assigns the resulting stub object to `el`.

2.4.2 Exception handling

When a Thor call signals an exception, the exception is given back to the client. In the C++ veneer, the client can provide a **handler**, which is a function that is called by the veneer when an exception occurs. If no handler is provided, the veneer terminates the client program with an error message.

2.4.3 Iterators

In Theta, iterators are called inside a **for** loop. the body of the loop and the iterator run as coroutines. On each iteration, the iterator is executed and yields a value. Control is then returned to the caller, which executes the loop body. The cycle repeats until the iterator terminates, or the loop is terminated prematurely using a **break** or **return** statement. For example, here is an example of a call to a Theta iterator:

```
for int: i in 1.to(100) do
  ...
end for
```

In this example, the iterator is `to`, which is defined on objects of the integer type, and yields all the integer values between the value of its receiver and the value of its first argument.

Currently, veneers do not provide a way for clients to call an iterator. In chapter 5, we present a new mechanism for calling iterators.

2.5 Discussion

The invocation mechanism described in this chapter is inefficient. Each call made by the client requires a round-trip communication with the Frontend, which is expensive. The next chapter describes a mechanism, **batched futures**[3], which reduces the overhead by amortizing the cost of a domain crossing across several calls which are sent to the Frontend in a **batch**. The remainder of the thesis describes two mechanisms for increasing the average batch size, and thus improving the overall performance.

Chapter 3

Batched Futures

Batched futures [3] is a mechanism that allows cross domain calls to the server to be batched. Batching calls improves performance, because the cost of a domain crossing between the client and the server is amortized over the number of calls in a batch. The greater the number of calls in the batch, the larger the performance benefit will be.

3.1 Batched Futures

The result of a call to a Thor operation is either a handle or a basic value. Because a handle can only be used as an argument to subsequent calls, delaying its computation makes sense. Instead of sending the call and waiting for the result handles to be computed, the call can be batched, and a placeholder generated for the result. At some later time, the batch of calls is sent to the Frontend and evaluated.

To allow future calls to refer to the result of the batched call, the stub for that call generates a place holder for the result, called a **future**. In addition to the handles for the arguments and the name of the method that is called, a future for the result of the call is included with the batched call. This will allow the Frontend to associate the result of the call with that future when the call is executed.

New calls are added to the batch as the corresponding stubs are invoked by the client program. When the client invokes a method that returns a basic value, the entire batch is sent to the Frontend and executed there. The Frontend executes each call, and assigns the result to the corresponding future. Because of the sequential nature of this process, futures are never used before they have been assigned a value.

For example, consider the following C++ code:

```
int nth(th_intlist *l, int n)
{
  for i = 1 to n do
    l = l->next();
  return l->first();
}
```

This function returns the nth element in a list of integers. It finds the nth node in the list by calling the `next` method n times, then returns the first element of the list.

Ordinarily, this function would require $n+1$ pairs of domain crossings: one pair for each invocation of the `next` function, plus one for the invocation of `first`. Using batched futures, however, this computation would require only a single pair of domain crossings. Each invocation of `next` returns a future for the result, which is used as the argument to the next call. As the calls are placed in the batch, the loop is essentially unrolled. Because `first` returns a basic value result, in this case an integer, its execution causes the batch to be sent to the Frontend.

The batch generated by the code in the previous example would look something like this:

```
f0 = h1->next()
f1 = f0->next()
f2 = f1->next()
...
return fn->first()
```

Each statement in the batch records all the information necessary for the Frontend to execute the call. For example, the first statement indicates that the `next` method should be invoked on the object with handle h_1. There are no arguments for this call, and the result should be assigned to future f_0. Similarly for the subsequent calls to `next`. Note that at each point in the batch, the future that is the receiver of the call has already been computed. The final call indicates that the result of calling the `first` method on future f_n should be returned to the client.

3.2 Implementation

This section contains a brief description of the batched futures implementation. A complete description of the implementation can be found in [3].

The batched futures implementation consists of the following parts:

- Modified veneer method-stubs that batch a message describing the call, and return a future for the result.
- A batch data structure that accumulates a sequence of calls, and is sent to the Frontend when a call is made that returns a basic value.
- A future table, F, at the Frontend, which maintains a mapping between futures and objects.
- A veneer future table, VF, which maintains a mapping between futures and stub objects.
- A counter, called the `future_index`, which is used to generate new futures.

3.2.1 Futures

Futures, like handles, are implemented as integer indices into a table of objects. To distinguish them from handles, futures are always negative. For the result of each batched call, a new future is generated by incrementing the `future_index`. This counter is periodically reset by the veneer as explained in the next section.

Each call placed in the batch contains the future assigned to its result. At the Frontend, the call is executed and the result is mapped to the specified future in the future table F. If a subsequent call presents this future as an argument, the corresponding object is fetched from F and passed as an argument to the actual call.

3.2.2 Future remapping

Future remapping is a mechanism used by the batched futures implementation to limit the size of the future table F. Because the veneer generates a fresh future for the result of each call, F could grow without bound. To prevent this, futures are periodically remapped to handles. During remapping, a new handle is created for each object in F by the Frontend. When all the futures have been assigned a handle, the Frontend sends these $< future, handle >$ pairs back to the veneer. Using the veneer future table, VF, the veneer finds the stub object corresponding to each future, and stores the new handle in it. The `future_index` is reset to zero, and all the space in both VF and F can be reclaimed.

Futures are remapped only when the `future_index` reaches a threshold value because it is a little more efficient to remap a large number of futures at once.

3.3 Discussion

Bogle's result showed that on a real benchmark, such as OO7[5], the average batch size (batching factor) is about 2.33, which is rather low. Consequently, the speedup achieved by using batched futures was only about 1.7. The remainder of this thesis describes two new mechanisms, promise and batched controls structures, which will enable a larger batching factor to be achieved.

Chapter 4

Promises

This section describes one of the two contributions of this thesis, **promises**. A promise is a reference for a basic value in Thor. Like regular references, it can be a future for the result of a call. In addition, a promise can be initialized to refer to a particular value. To get the actual value the promise refers to, the promise is **claimed**.

Promises were first described in Mercury call streams [13]. However, a Mercury promise must be claimed before it can be used as an argument to a subsequent call. A Thor promise does not have to be claimed before it is used as an argument. It only needs to be claimed if the actual value of the promise is needed.

Promises can be used to increase the batching factor. If a basic value result is not of immediate interest to the client, the client can request that the call return a promise rather than an actual value. This allows the call to be batched, using the batching mechanism described in section 3.1.

Although promises can be used to improve the batching factor in certain cases, in practice we expect their benefit to be relatively small. However, they are necessary for Batched Control Structures (chapter 5), which we expect to have a much larger impact on the batching factor.

4.1 Improving the Batching Factor using Promises

Sometimes, basic value results are used in the same ways as are handles, i.e. they are passed as arguments to later calls, but their value is never used directly. In those cases, it would make sense to delay the computation of the actual value. For example, if the client program exchanges two integer fields in an object, there is no need to examine the actual value of the fields. Without promises, the code might look something like this (in C++):

```
swap(th_point *p) {
  int t1 = p->x();             // Get val of x field
  int t2 = p->y();             // Get val of y field
  p->set_x(t2);                // Set x field to old y
  p->set_y(t1);                // Set y field to old x
}
```

In this example, p is a stub object pointer of the type th_point, which corresponds to the point type in Thor. Point is a mutable type, with methods `x` and `y` to get the corresponding values, and `set_x` and `set_y` to assign new value to them.

The method calls `p->x()` and `p->y()` cannot be batched (because they both return integers). Thus this section of code incurs the expense of three pairs of domain crossings: one each for the calls to `x` and `y`, and one for the batch containing the calls to `set_x` and `set_y`.

With promises, the same code requires only a single pair of domain crossings. Rather than calling `p->x()` and `p->y()`, which return integers, special stub functions are called that return integer promises instead. In the following example these functions have the first letter of their name capitalized:

```
swap(th_point *p) {
  th_int *t1 = p->X();          // Assign x field to a promise
  th_int *t2 = p->Y();          // Assign y field to a promise
  p->Set_x(t2);                 // Set x field to old y
  p->Set_y(t1);                 // Set y field to old x
}
```

Note that `X` and `Y` return not integers, but stub objects of type th_int. Similarly, `Set_x` and `Set_y` take th_int's as arguments instead of C++ integers. Th_int corresponds to the Thor integer type, except that it supports an additional **claim** method. In this example, the promises are never claimed. However, as another example, a promise could be used to keep a running sum, in which case it would be claimed when the sum had been computed.

4.2 Implementation at the C++ Veneer

At the veneer, a promise is implemented as a tagged union of a future and a value. The value can be set explicitly when a promise is created, or it can be set by the veneer when the actual value of the promise becomes available.

Promises are encapsulated in stub objects like all other Thor references. The stub object types for promises have a special claim method that is used to obtain the actual value of the promise. Unlike other stub objects, a promise can only contain a future, never a handle. This is because a call that returns a promises is always batched, which means that the stub object returned by such a call contains a future. The future is replaced with a value, rather than a handle, during future remapping.

4.2.1 Method Stubs

C++ is a strongly typed language. Since a promise and its corresponding basic value are different types in C++, the veneer cannot use a single stub function that accepts either a basic value or a promise for a particular argument. Similarly, the same stub function cannot be used to return either a basic value or a promise for the result. Thus, the veneer provides two different versions for each Thor routine or method that has basic value arguments/return values, one that takes or returns promises, and one that takes or returns basic values. This will at most double the number of stub functions that must be provided. Since many methods do not take or return basic values, the actual expansion factor is usually much less. In the current implementation, it's roughly 1.5.

As an example, here is the new definition for the th_point class used in the previous example, with support for promises:

```
class th_point: public th_any {
 public:
  int x();                  // return value of x field
  int y();                  // return value of y field
  void set_x(int x);        // set value of x field
  void set_y(int x);        // set value of y field

  th_int *X();              // return promise for value of x field
  th_int *Y();              // return promise for value of y field
  void Set_x(th_int *x);    // set value of x field with a promise
  void Set_y(th_int *y);    // set value of y field with a promise
}
```

The new definition contains 4 additional methods to support promises: `X` and `Y`, which return integer promises, and `Set_x` and `Set_y`, which take integer promises as arguments.

In some cases, the client programmer may want to call a Thor method with a mixture of promises and basic value arguments/return values. There are three alternatives for providing this functionality:

1. Provide a different stub for each permutation of promises and basic value arguments and return values.
2. Provide a mechanism for converting basic values into promises so that the promises version can be used.
3. Require that all promise arguments be claimed first, so that the basic value version can be used.

The first option, providing a different stub function for each permutation of promises and basic value arguments and return values, would lead to an exponential increase in the amount of code provided. So, instead, we chose a combination of the second and third options. Actually the third option exists by default, but does not provide the full functionality desired, because it does not allow the client to call a stub that returns a promise when a mixture of promise/value arguments is to be used. It may also defeat the purpose of providing promises in the first place by forcing a promise to be claimed unnecessarily.

To convert a basic value into a promise, the veneer provides a constructor that can be used to create a new promise with a given value. In the C++ veneer, the constructor is defined as part of the stub object's class. Thus, to create an integer promise with value 5, the code would be:

```
th_int* foo = new th_int(5);
```

The constructor simply sets the value field of the promise, and sets the tag accordingly.

The complete specification for a th_int is as follows. The definitions for th_bool, th_real, and th_char, which correspond to Theta boolean, real number and character values are similar:

```
class th_int : public th_any  // future is stored in th_any's handle field.
{
 public:

   // Stubs for regular Theta integer methods

   int add(int y);
   int sub(int y);
   int mul(int y);
   int div(int y);
       /* signals(zero) */
   int mod(int y);
       /* signals(zero) */

   bool equal(int y);
   bool lt(int y);        // Less than
   bool gt(int y);        // Greater than
   bool le(int y);        // Less than or equal to
   bool ge(int y);        // Greater than or equal to
   ...

   int to(int y);         // Iterator (eg. for i in 1.to(5) do)

   // Method stubs that support promises

   th_int(int val_)       // Sets the value field and tag.

   th_int* Add(th_int* y);
   th_int* Sub(th_int* y);
   th_int* Mul(th_int* y);
   th_int* Div(th_int* y);
       /* signals(zero) */
   th_int* Mod(th_int* y);
       /* signals(zero) */

   th_bool* Equal(th_int* y);
   th_bool* Lt(th_int* y);
   th_bool* Gt(th_int* y);
   th_bool* Le(th_int* y);
   th_bool* Ge(th_int* y);

   th_int* To(th_int* y);

   int claim();           // Extract actual value.
```

```
 private:
   int val;              // Value field of promise
};
```

4.2.2 Batching a call with promises

As with regular futures, a stub function that returns a promise generates a future for the result, and creates a new stub object containing that future. The future is included with the call so that the Frontend can assign the result of the call to that future. The stub then returns a pointer to the newly created stub object. A promise argument is sent to the Frontend either as a future or as a value, with a tag to distinguish the two.

4.2.3 Future Remapping and Claim

To limit the size of the future tables, the veneer periodically remaps futures to handles (section 3.2.2). Originally, remapping was done after the `future_index` value had exceeded some threshold. This allowed the remappings to be processed in larger chunks, as opposed to remapping only the futures used in a particular batch, which tended to be small. However, this policy was changed when promises were implemented, so that remapping is done after the execution of every batch. This was done for two reasons:

1. We expect the average batch size to be larger with the use of promises and Batched Control Structures.
2. The claim operation can be implemented very efficiently.

During remapping, a future for a basic value is remapped to the actual value rather than to a handle. This value is stored in the value field of the promise, and the tag is changed accordingly. When a promise is claimed, and its tag indicates that it contains a future, the current batch is sent to the Frontend. All futures are remapped after the batch has finished executing. Thus all the promise stub objects contain the actual value of the promise after a batch has finished executing.

4.3 Implementation at the Frontend

The Frontend receives a promise as either a value or a future. If the promise is a value, the Frontend treats it as it would treat a normal basic value argument. If it is a future, the corresponding value is looked up in the future table F, and used as the argument to the actual call.

4.4 Experimental Results

In [3], Bogle uses his performance model to describe the expected impact of promises on the performance of the 2b traversal of the OO7 benchmark[5]. After implementing promises, I confirmed his predicted result.

4.4.1 Performance Model

Bogle's model states that the average cost of a call is

$t_c + t_d/B$

where t_c is the average cost of running a call, t_d is the cost of a pair of domain crossings, and B is the "batching factor" i.e. the total number of calls divided by the number of pairs of domain crossings.

4.4.2 OO7 Benchmark revisited

Bogle predicted that using promises in the 2b traversal would increase the batching factor from 2.33 to 3.47. This prediction was based on replacing the use of integers with promises in a part of the traversal code similar to the examples discussed earlier, where two integer fields are swapped without their value being examined. Rewriting the code as suggested by Bogle yielded exactly the predicted batching factor of 3.47. However, since Bogle's measurements, the Thor invocation mechanism has been optimized, yielding about a factor of 2 improvement in performance[6]. The domain crossing overhead, t_d has been substantially decreased, thus reducing the total benefit achieved from increased batching.

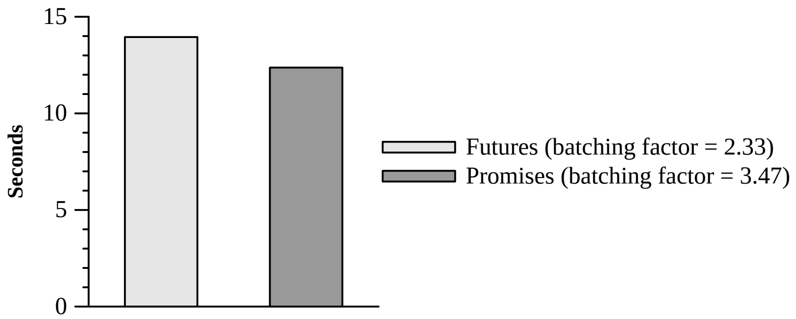

Figure 4-1: Promises in OO7 2b traversal.

Figure 4-1 is a comparison of the running time of the OO7 2b traversal using batched futures versus the running time of the same traversal modified to use promises as well. The running time for the batched futures version was 13.95 seconds. The running time for the promises version was 12.37 seconds. Thus the speedup achieved was about 11%.

Chapter 5

Batched Control Structures

This chapter describes the main contribution of this thesis, **Batched Control Structures (BCS)**. BCS is a mechanism that allows a client program to send small "programs" to the server to be executed there. BCS allows the client programmer to express simple control flow such as if statements and loops in a batch. This has several advantages:

- Using BCS can increase the number of calls that can be placed in a batch.
- Using BCS amortizes type checking and other call overhead at the Frontend across multiple iterations of the same code.
- BCS provides a mechanism to call iterators.

5.1 Increasing the Batching Factor using BCS and Promises

In some applications, the batching factor is low due to the structure of the client program. For example, a client loop with a condition that depends on a Thor call will cause a new batch to be sent and evaluated on each iteration of the loop because the call has to be evaluated on each iteration so that the control flow of the loop can be determined. BCS can solve this problem by allowing the client to batch the entire loop, thus reducing the number of domain crossings from 1 per iteration to at most 1 for the entire loop.

Specifically, let's consider a while loop that traverses a Thor list, where the condition is a single call to the Frontend that returns a boolean value:

```
th_list *l;

while (l->notEmpty()) { // Single call returns a boolean
  ...
  ...                   // n calls here
}
```

If the loop body contains n calls, and the loop executes k iterations, the client, using batched futures, will send k batches of size $n+1$ to the Frontend, each requiring a round-trip communication. The extra call in the batch accounts for the call that computes the loop condition, a->notEmpty(). This call causes the batch to be sent to the Frontend on each iteration because it is a call that returns a basic value, in this case a boolean.

If the entire loop could somehow be sent to the Frontend, and executed there, the effective batch size for this piece of code would be the number of iterations the loop executes at the Frontend times the number of calls in the loop, i.e. $k(n+1)$. Thus, the entire loop over all its iterations incurs the expense of only a single round-trip communication.

Assuming that batching the entire control structure does not significantly increase the overhead of evaluating the batch, this increase in the batch size should result in an overall increase in performance. In Chapter 7 we will show that after only a few iterations, the cost of processing a batched control structure is successfully amortized over the number of times it is evaluated.

Sending an entire loop across to the Frontend requires that:

- Calls that return basic values can be batched.
- The client can express control flow statements such as while and if in the batch sent to the Frontend.
- The Frontend can reconstruct the control structure from the information in the batch and execute it.

To satisfy the first requirement, promises can be used. The second and third requirements are addressed by the BCS mechanism itself, which is described in more detail in the remainder of this chapter.

In theory, the client would be able to send arbitrarily complex code to the Frontend for evaluation. However, the more complex the code, the more difficult it is to parse and evaluate it efficiently. In addition, we do not want the client programmer to have to learn a complex new language. Instead, the goal of BCS is to provide the programmer with a small number of useful constructs that can be used to improve the performance of an application. Specifically, this work will be concerned only with the following control structures:

- if statements
- while loops
- for loops

We don't expect batched if statements used in isolation to provide an improvement in performance, because the statements inside the if control structure are executed at most once. Thus the cost of parsing a batched if control structure cannot be amortized across multiple iterations. However, when an if control structure is nested inside a batched while or for control structure, parsing of the if control structure does get amortized across the number of iterations of the enclosing loop. This is the intended use of batched if control structures.

5.2 BCS syntax and semantics

The BCS mechanism is a direct extension of the batched futures mechanism described in Chapter 3. Batched futures allows the batching of a sequence of calls to Thor methods and routines.

The BCS mechanism extends the veneer with a set of control flow statements that can be used explicitly by the client program to batch a control structure. Execution of these statements annotates the batch with begin and end markers to delimit the control structure. The body of the control structure must be made up entirely of calls to Thor routines or methods, which are batched using batched futures and promises.

As the batch is parsed by the Frontend, the markers are used to reconstruct the control structure, which is then evaluated. The evaluation process is described in detail in Chapter 6.

The remainder of this section describes the syntax and semantics of BCS. In this section we use a description language based on the C++ implementation of BCS. However, BCS is a general mechanism that does not rely on any specific C++ language features. The syntax and semantics for BCS would be similar in other languages. This issue is discussed in more detail in Chapter 6.

5.2.1 BCS grammar

The reference grammar for the BCS language is as follows:

BCS → WHILE(expr) body ENDWHILE |

IF(expr) body ELSE body ENDIF |

FOREACH(stub_pointer, batched_method_call) body ENDFOREACH

expr → batched_method_call | stub_pointer

batched_method_call → expr − > method_name (expr [, expr] ∗)

body → [method_call | BCS] ∗

In this grammar, stub_pointer is a client identifier that refers to a stub-object, and batched_method_call is a method call that can be batched, i.e., it does not return any basic value results.

5.2.2 Batched While loop

The argument to the WHILE call is a boolean expression that is evaluated at the Frontend on each iteration of the loop. The body of the loop executes as long as the expression evaluates to TRUE. For example, the list traversing loop presented earlier would be rewritten as follows:

```
th_list *l;

WHILE (l->NotEmpty()) // Single call returns boolean promise
  ...
  ...                 // n calls
END_WHILE
```

5.2.3 Batched If statement

The semantics of IF are very straightforward. The condition must result in a boolean promise. At the Frontend, the condition is evaluated. If the resulting value corresponds to

TRUE, the if-body is evaluated; otherwise, the else-body is evaluated. ELSE is optional of course.

5.2.4 Batched For loop

A batched for loop provides a very efficient mechanism for calling an iterator. In the FOREACH statement, the `batched_method_call` must be a single call to an iterator. In the scope of the body, `stub_pointer` refers to the value yielded by the iterator. In general, iterators can yield multiple values on each iteration. However the FOREACH construct deals only with the common case of an iterator yielding a single value. To support yielding of multiple values, the FOREACH construct can easily be extended by taking a sequence of stub_pointers instead of a single stub_pointer in the first argument. Here's an example of a for loop that iterates over the integers 1 through 100:

```
th_int *one = new th_int(1);
th_int *one_hundred = new th_int(100);
th_int *i;

FOREACH(i, one->To(one_hundred))
  a->foo(i);
ENDFOR
```

In the scope of the body of the FOREACH statement, the variable `i` refers to the value returned by the iterator `To`. The values 1 and 100 have to be converted into promises because they will be used as arguments to a call that returns a promise (`To`), as explained in section 4.2.1.

When evaluated by the server, the body and the iterator execute as co-routines. The iterator executes once and the result is assigned to i. Then the body is executed once. This continues until the iterator signals that there are no more values to be yielded. The current implementation does not provide a means for terminating an iterator before all the values have been yielded. This is important, because some iterators may be non-terminating (e.g. an iterator that yields prime numbers). Chapter 8 describes a mechanism for terminating a batched while or for loop using an analog of the `break` statement in C.

5.2.5 Assignment

Both the syntax and semantics of BCS are designed to mimic the corresponding language constructs in Theta[8]. However, unlike Theta, the BCS language is not a compiled language in its own right, but instead is implemented as a bunch of procedures that can be called by the client to annotate the batch (Chapter 6). This places certain limitations on how the BCS mechanism can be used.

First, because the BEGIN and END statements are procedure calls that place markers into the current batch, they do not interrupt the sequential flow of control in the client program. Thus, all the statements following the BEGIN and preceding the END are executed as they would have been in the absence of the BCS commands. Only statements that cause things to be placed into the batch will contribute to the body of the batched control structure as seen by the Frontend. These include calls to Thor method stubs, and nested BCS statements. Thus, any calls to client procedures, or assignments to client variables, will not be expressed in the batch sent to the Frontend. Note that this means that such calls or assignments will be executed exactly once at the client, even though at the Frontend the containing control structure may be evaluated zero or more times.

Second, the batched futures mechanism was designed to batch a linear sequence of calls. As each call is batched a future is generated for the result. When the call is executed by the Frontend, the future for the result is mapped to the actual result in the future table F. When a later call uses the future as an argument, the object is looked up in F and passed to the actual call. The client uses stub objects to refer to futures. Typically, the stub object returned by a method stub is assigned to a client identifier, which now refers to the future for the result of the batched call. Wherever that identifier is used as an argument, the corresponding future is used in the batched call. Thus there is always a one-to-one relationship between the identifier and the future.

However, when an assignment is made to an identifier in the body of a batched control structure, this one-to-one relationship may be violated. For example, consider the following code fragment:

```
th_list *l = find_list();

WHILE(l->NotEmpty())
x = l->first();
...
l = l->next();
END_WHILE
```

In the first statement, `l` is assigned the result of a call to a client procedure that returns a pointer to a list stub object. Thus `l` refers to some future, f_i, and in both the condition of the WHILE statement, and in the call to `first`, f_i is used in the batched call as the receiver of the call. Also, in the last statement in the body, f_i is used at the receiver of the call (to `next`). The result of this call corresponds to a different future, f_j, which is assigned to `l`. However, there is no way to propagate this information back to the top of the WHILE loop. All the calls involving `l` have already been batched, using f_i as the future. When the loop is evaluated by the Frontend, this same future is used on each iteration, and the loop never terminates, because the value of `l` never changes.

The problem here is that there is no longer a one-to-one correspondence between `l` and the result of a particular call. Instead, `l` refers to f_i after the first assignment, and to f_j after the second assignment. We expect `l` to refer to the result of the first assignment on the first iteration of the loop, and to the second assignment on the second iteration, etc. However, this is not what happens.

Assignment to an identifier inside a batched control structure will work as expected provided the assignment does not occur in a statement inside the body that textually follows a use of that identifier in the body or condition of the control structure. Thus, in our previous example, the assignment and subsequent use of `x` will have the expected behavior because `x` is not used in the control structure prior to the assignment to `x`. However, this is not true for `l`, because `l` is used in the condition, and then assigned to in the body. It is clear from the example, however, that there are common situations in which it is necessary to use an identifier before it is assigned to inside the body of a batched loop. A mechanism that provides this functionality is described below.

Another question raised by assignment to an identifier inside a batched control structure (e.g. `x` in the previous example) is: what Thor object does such an identifier refer to after the control structure has been evaluated? A reasonable semantics might be that the identifier refers to the result of the last time that call was evaluated. However, it is possible that the call is never evaluated at all, since the body of a batched while loop, or the branch of a batched if statement is conditionally evaluated. So, an identifier assigned to inside a batched control structure does not necessarily have a well-defined meaning after the control structure has been evaluated. In fact, in the current implementation, such an identifier *never* refers to a valid Thor object after the control structure has been evaluated.

The solution we settled on for both of these issues was to extend Thor slightly by inventing a new type called a **cell**. An object of type cell stores a single value of some type. Assignment to an identifier for a stub object inside a control structure can be rewritten as performing storage operations on an object of type cell. The next section gives a more detailed explanation of cells and how they are used.

Note that the BCS mechanism cannot detect an inappropriate assignment inside the body of a batched control structure, or an inappropriate use of an identifier after the batched control structure. It is up to the programmer to use cells in the appropriate situations. Of course, it is always safe to replace assignment with the use of a cell. However, it is not always necessary, and may slightly increase the overhead of the batched control structure.

5.2.6 Cells

The idea behind cells is to replace assignment to a client variable with a method call on a Thor object. One advantage of this approach is that the semantics of method calls inside a batched control structure is well understood, thus making it easier to reason about the resulting program. Also, unlike other potential solutions, cells do not increase the complexity of the veneer or the BCS mechanism.

The Theta specification of a cell is as follows:

cell = **type**[T]

  put(v : T)
    % Store a new value in the cell.

  get() **returns**(T), **signals**(not_possible)
    % Return value stored by previous put call.
    % Signal if no value has been stored in this cell.

The `Cell` type is parameretized so that information about the type of the value stored in the cell is preserved. This greatly simplifies the use of a cell.

Prior to the control structure, the client creates a new cell by calling a special veneer function called `th_cell_new`. In C++ this function is defined as a constructor for the `th_cell` class. A call to `th_cell_new` batches a special command to the Frontend that creates a new cell, and then returns a future for the cell. Assignment is replaced by a call to the put method on the cell, which mutates the cell by storing the new value. To retrieve the value that was assigned to the cell, get is called. For example, the list traversal code presented in the previous example would be rewritten as follows:

```
th_list *l = ...;
th_cell<th_list> *cl = new th_cell<th_list>;  // create new cell

cl->put(l);
WHILE(cl->get()->NotEmpty())
  x = cl->get()->first();
  ...
  cl->put(cl->get()->next());
END_WHILE
```

In this example, a new cell is created with the call to new th_cell. The cell is initialized with a call to put, with the list l as the argument. Inside the loop body, any access to l is replaced with a call to get. Finally, the last statement stores the new value of the list in the cell.

Note that in this example, cl can be used to obtain the result of the last call to next after the batched control structure has been evaluated.

5.2.7 When to use BCS

Chapter 7 will show that it is possible to get a substantial improvement in performance by replacing a client loop with a batched control structure, even for a small number of iterations. Such replacement may place somewhat of a burden on the client programmer however. In order to batch the condition of a while loop or if statement, the condition must be rewritten entirely in terms of calls to Thor methods or routines. If the body of the loop contains calls to client procedures, it may not make sense to use the BCS mechanism, since that would require somehow eliminating those calls or rewriting them in terms of Thor calls. Generally, BCS should be used whenever it is relatively simple to convert client code to BCS.

The following is an example of how a client loop is converted to a batched loop, specifically how to change the condition. Rewriting the condition may slightly increase its complexity, because client language operators on basic values have to be replaced with equivalent Thor calls, and basic values have to be replaced with basic value promises. The first is necessary so that the condition can be computed by the Frontend, and the second is necessary so that the calls in the condition can be batched. For example, consider the following while loop in C++:

```
th_list *l;

while (l->length() < 100) {   // Single Thor call: l->length()
  ...
}
```

Rewriting the condition requires converting the $<$ operation to the `Lt` method defined on integer promises, and replacing the integer value 100 with an integer promise initialized to 100. Specifically, the code would look like this:

```
th_list *l;
th_int *limit = new th_int(100);    // Initialize integer promise

WHILE (l->Length()->Lt(limit))   // Condition with only Thor calls.
  ...
END_WHILE
```

`Lt` returns a boolean promise, whereas `lt` would return a boolean.

5.2.8 Exception handling

Giving the client program access to an exception that occurs during the evaluation of a batched control structure is difficult, because there is no simple way to associate the exception with the actual call that generated it. Thus, in the current implementation, an exception generated during the evaluation of a control structure causes the client program to terminate with an error message. It is intended that BCS be used to improve the performance of well-tested code that is not likely to signal any exceptions. Exceptions signaled by the BCS mechanism itself (as explained in the next chapter), are analogous to the syntax errors generated by a compiler.

Chapter 6

BCS Implementation

This chapter describes the implementation of Batched Control Structures. The implementation consists of three parts: First, the language used by the veneer and the Frontend to communicate control flow information in the batch; second, the implementation of the BCS syntax described in the previous chapter; third, the mechanisms at the Frontend that interpret and evaluate the batched control structures.

6.1 Annotated Batch

Batched control structures are communicated to the Frontend by annotating the batch with control flow information. For example, this while loop:

```
th_int *limit = new th_int(100);

WHILE (cl->get()->Length()->Lt(limit))
  ...
  cl->put(cl->get()->next());
END_WHILE
```

would produce the following annotated batch:

```
BEGIN_WHILE_COND                   // Beginning of condition expression
f1 := f0->get()
f2 := f1->Length()
f3 := f2->Lt(100)                  // limit is a promise with value 100
BEGIN_WHILE_BODY f3                // End of expression; begin body
...
f4 := f0->get()
f5 := f4->next()
f0->put(f5)
END_WHILE_BODY                     // End of body.
```

The BEGIN_WHILE_COND marker indicates the beginning of the expression for the condition of the loop. The calls in the expression are batched as before. The BEGIN_WHILE_BODY marker indicates the end of the condition, and the beginning of the body. The future f3 is sent along with this marker. This future refers to the boolean value to be checked as the condition. The calls in the body are batched as before, and any nested control structures are batched in the same way as this one. Finally, the END_WHILE_BODY marker indicates the end of the batched loop. Similar markers are provided for the IF and FOREACH control structures.

6.2 BCS language implementation in C++

In the C++ veneer, the BCS commands (e.g WHILE, ENDWHILE, FOREACH, ENDFOREACH, IF, ELSE and ENDIF), are implemented as macros. Each macro expands to a sequence of procedure calls that place the appropriate markers in the batch. The IF and WHILE macros take the condition expression as their single argument, and place markers around it in the batch. The FOREACH macro takes two arguments, the stub object that holds the value yielded by the iterator, and the expression for the iterator.

For example, the WHILE(cond) macro expands to the following pseudocode:

```
put_begin_while();              // Place BEGIN_WHILE_COND marker in batch
th_bool e = cond;               // Batch condition expression
put_begin_while_body(e);        // Place BEGIN_WHILE_BODY marker in batch
```

The assignment of the condition to the boolean promise `e` causes the calls in the condition to be placed in the batch when this code is executed. The promise is then given as an argument to the `put_begin_while_body` function. This function places the future referred to by `e` in the batch following the BEGIN_BODY marker. Note also that because the condition is assigned to a boolean promise, the compiler will type check the call to ensure that the expression does indeed evaluate to a boolean promise. The IF macro is exactly the same as the WHILE macro, except that it calls `put_begin_if`, etc.

In the case of the FOREACH(var, iter) command, the variable is assigned the result of calling the iterator. Thus, the statement `FOREACH(i, one->to(hundred))` expands to:

```
put_begin_foreach();            // Place BEGIN_FOREACH marker in batch
i = one->to(hundred);           // Batch a call to the iterator
```

Again, the assignment of the iterator to the variable is type checked by the compiler. The assignment generates a future for the value yielded by the iterator, which is stored in the stub object referred to by the variable `i`. Anywhere inside the body where `i` is used as an argument or receiver of a call, that future will be used in the batch. When the loop is evaluated by the Frontend, this future refers to the current value yielded by the iterator on each iteration. No marker is needed to indicate the beginning of the body of a FOREACH control structure because the iter argument must expand to a single call. Thus the body begins with the second call after the BEGIN_FOREACH marker.

The ELSE and ENDIF, ENDWHILE and ENDFOREACH all expand to a single call that places the appropriate marker in the batch.

Note that the expression for the condition in the case of WHILE and IF, is surrounded by two markers. One advantage of using macros (as opposed to regular procedure calls) is that the syntax hides this level of detail from the programmer. When the macro is expanded, the cond expression is not evaluated. Thus the macro can sandwich the expression between the calls that place the markers, so that when the code is executed, the calls in the expression will be batched after the BEGIN_EXPR marker, and before the BEGIN_BODY marker. In a language that does not support macro processing, the client programmer would have to do this marker placement explicitly, by essentially hand-coding the macros presented above.

One shortcoming of the current implementation is that the following two syntax errors are not checked at compile time. First, the compiler cannot check whether the BEGIN and END markers match. So, for example, a WHILE statement can be ended by an ENDIF, or ENDFOREACH. The BCS parser at the Frontend does check for this error, at run-time. Second, the compiler *can* verify whether the WHILE, IF or FOREACH statement is terminated at all, because the actual macro introduces a new scope, which is terminated by any END statement. However, this does not ensure that the appropriate END marker has been placed in the batch, as the scope can be accidentally terminated by other means, without using the END macro. The macros could be modified so that they encorporate the entire control structure, in which case the compiler would be able to check for both of these errors. Thus, for example, the WHILE(cond) macro could be changed to WHILE(condition, body). The ENDWHILE macro would no longer be necessary, as its functionality would be encorporated in the WHILE macro. The disadvantage of this approach is that the syntax would be slightly less natural in C++. Also, the fact that a control structure is not terminated cannot be detected by the Frontend (as explained in the next section), so this problem would remain for languages that cannot use macros to implement BCS.

Since the veneer uses the batched futures mechanism to batch Thor operations inside a batched control structure, no further extensions are needed to the veneer to support BCS. All the work of evaluating the control structure is done by the Frontend.

6.3 BCS Implementation at the Frontend

When the Frontend parses the batch, stand alone method calls are parsed and executed as before. When a BCS begin marker is reached, calls are processed differently. Each call is type checked and added to a parse tree node representing the batched control structure. Control structures can be nested. When the end marker for the outer most control structure is reached, the Frontend evaluates the entire parse tree. This process continues as long as there are statements in the batch.

6.3.1 Syntax checking

While parsing a batch, the Frontend can detect a begin-end marker mismatch, e.g. if it finds an END_IF marker when the current control structure is a while loop, it signals an exception. However, the Frontend cannot detect that the batch does not contain an END marker at all for the current control structure. This is because the batch is implemented as a continuous stream that does not have a fixed end. This approach allows the client and the

Frontend to place things in and take things out of the batch asynchronously whenever they are scheduled to execute by the operating system. Thus, if the client fails to provide an END marker for a control structure, the Frontend will continue processing subsequent batched calls and control structures as if they were nested inside the current control structure. But, if the client makes a call that cannot be batched, before terminating a batched control structure, the Frontend does signal an exception. Since the client will almost certainly make such a call in order to get some basic value results, the error will eventually be detected.

6.3.2 Type checking

The goal of type checking a call is to ensure that, at run-time, an object passed as an argument to the call will be a subtype of the type listed in the method's signature for that particular argument.

In the current system, this guarantee is checked for each method call as it is submitted by the client. However, to improve the performance of BCS, the method calls in a batched control structures are type checked before the control structure is evaluated. This amortizes the overhead of type checking over the number of times the control structure is evaluated.

In this discussion, the following notation will be used:

- **Apparent type (A)**: The type of an object as inferred by the type checking system. For argument, x, $A(x)$ denotes the apparent type of x.
- **Actual type (C)**: The real type of the object; this information is contained in the object. For argument, x, $C(x)$ denotes the actual type of x.
- **Expected type (E)**: The type listed in the signature of a method for one of its arguments. For argument, x, $E(x)$ denotes the expected type of x.
- $S < T$: S is a subtype of T.
- **constant future**: During the evaluation of a batched control structure, a constant future refers to the same object on each iteration (i.e. the future refers to the result of a call made prior to the beginning of the control structure).
- **variable future**: During the evaluation of a batched control structure, a variable future can refer to a different object on each iteration (i.e. the future refers to the result of a call made inside the control structure).

Type checking algorithm

A batched method call inside a control structure consists of the following items: a reference or value for the receiver of the call, the name of the method, references or values for the arguments, and a future for the result.

To ensure that a call is type correct, the type checking algorithm has to verify that for each argument (including the receiver), a, $C(a) < E(a)$. $E(a)$ is determined by looking at the method signature. The check whether $C(a) < E(a)$ has to be done differently, depending on whether a is referred to by a constant reference (handle or constant future), or a variable future. Recall that basic value arguments do not need to be type checked.

If a is referred to by a handle or a constant future, $C(a)$ can be determined directly from the object, because the object will be in the handle or future table. Thus the algorithm can verify that $C(a) < E(a)$.

```
BEGIN_WHILE_BODY
...
f_i = f_j->get()                // get() returns a list.
...
f_i->first()                    // A(f_i) = list, E(f_i) = list
...
END_WHILE_BODY
```

If a is referred to by a variable future (e.g. f_i in the above example), $C(a)$ cannot be determined because the object is not available at parse time. However, f_i refers to the result of an earlier call to a method, m_i. $A(a)$ is defined as the return type listed in the signature of m_i. Theta methods are type safe, which guarantees that the type of the object returned by the method will be a subtype of the type listed in the method signature for the return value. Thus, $C(a) < A(a)$. Because the subtype relation is transitive, the algorithm need only check that $A(a) < E(a)$ to ensure that $C(a)$ will always be a subtype of $E(a)$.

To determine the apparent type of an object referred to by a variable future, f_v, a mapping is created between f_v and the return type of the method, m_i, whose result f_v refers to. Thus, in the previous example, when the call to `get` is parsed, a mapping is created between f_i and the type `list`. These mappings are kept in a table, called T for **type table**. When f_v is later presented as an argument to a call, the apparent type of the corresponding object is determined by looking it up in T. Thus, in the previous example, when the call to `first` is type checked, the apparent type of f_i is found by looking up f_i in T.

6.3.3 Offset computation

At run-time, before an argument object is passed to a method call, its representation has to be changed so that it will appear to the method as an object of the type expected by the method. This is done by computing an integer **offset** that depends on the relationship in the type hierarchy between the object's actual type and the type expected by the method. A complete description of offsets and their use can be found in [14].

For each call sent by the client, the offset has to be computed for all the arguments before the method is called. In the common case the actual type and the expected type are identical, and computing the offset is trivial. However, when the two are not identical, computing the offset requires traversing the type hierarchy starting with the actual type and examining all supertypes recursively until the expected type has been reached. During the type checking stage of BCS evaluation, part of the cost of computing the offset can be amortized across the number of times the same method will be called inside a control structure.

The actual type of an object cannot be known until the object has been computed. Thus the complete offset between the actual type and the expected type cannot be computed during type checking for a variable future. For a constant future, or handle, it can be computed, and this information is stored in the parse tree. What *can* be computed for a variable future during type checking, however, is the offset between the apparent type and the expected type. This information is also stored in the parse tree. Prior to executing the method call, the offset between the actual type and the apparent type is computed and added to the previously computed offset. This sum constitutes the total offset between the actual type and the expected type.

6.4 Parse Tree construction and evaluation

As the Frontend parses the batch, it processes normal calls and executes them as described in Chapter 2. When a begin marker for a batched control structure is reached, the Frontend changes its mode. A parse tree is created to represent the control structure. Subsequent calls and batched control structures are added to the parse tree. When the end marker for the outermost control structure is reached, the parse tree is evaluated. Processing of the batch then continues as before. This section describes the structure of the parse tree and how it is evaluated.

6.4.1 Parse Tree Nodes

A BCS parse tree is constructed from Nodes of the following types: IF, WHILE, FOREACH, Method and Arg. IF, WHILE and FOREACH Nodes may contain nested BCS parse trees. A Method Node represents a particular method call, and includes the method, the receiver and arguments, and where to place the results. The IF, WHILE, and FOREACH nodes represent the corresponding control constructs. An Arg Node is used to represent an argument to a method call.

After the parse tree has been constructed, it is evaluated. Evaluation starts by calling the eval method of the top-level parse tree Node. This function recursively calls the eval methods on any parse trees this Node contains. If the parse tree is a list of Nodes, the eval method is called on each member of the list. Evaluation halts when there are no more Nodes to be evaluated. In the current implementation, if an exception is signaled during BCS parsing or evaluation, the connection between Frontend and client is terminated.

6.4.2 Arg Nodes

An Arg Node is used to store information about a particular argument to a call. Arg Nodes can be divided into three different types:

1. If the argument to a call is referred to by a handle, or a constant future, the corresponding Arg Node contains a pointer to the object; this pointer is obtained from the handle or future table at parse time. This type of Arg Node is referred to as a Const Arg Node.

2. If the argument is a basic value, the value is stored directly in the Arg Node. This is called a Basic Arg Node.

3. In the case of a variable future, a special kind of Arg Node is used, called a Var Arg Node.

Each of these Arg Node types is described in more detail in the remainder of this section.

Const and Basic Arg Nodes

A Const Arg Nodes contains a pointer to the object. This pointer has already been offset properly. A Basic Arg Node contains the basic value itself. In both cases, when evaluated, the Arg Node simply returns its value. Both of these Arg Nodes are very simple. As explained in section 6.4.3, they are often eliminated during Method Node construction and optimization.

Var Arg Nodes

A future, f_c, refers to the result of a particular call. When a later call contains f_c in its argument list, the Frontend looks up f_c in the future table to find the corresponding object, which is then passed to the actual call.

The same approach could be used for calls inside a batched control structure. However, in a batched control structure, the same sequence of calls is executed many times, which means that the same futures would be looked-up in F many times. Instead, it makes more sense to use a mechanism that does a little more work at parse time, but incurs less overhead at evaluation time.

Instead of using the future table to pass results of earlier calls to later ones, the BCS evaluation mechanism uses shared pointers to pass the results. For each variable future, f_v, a pointer object, p_r, is used that at any given time contains a pointer to the object referred to by f_v. f_v refers to the result of some method call. When this call is evaluated, p_r is mutated so that it points to this result. Whenever a call uses f_v as an argument, it dereferences p_r to get the corresponding object.

All the Arg Nodes that correspond to f_v contain a pointer to p_r. Each Arg Node also contains the partial offset (section 6.3.3), and the apparent type of the object so that the full offset can be computed during evaluation. The partial offset depends on the expected type of the argument, and thus may be different for different method calls that take the object referred to by f_v as an argument.

When the Arg Node is evaluated, it verifies that p_r contains a valid object pointer (p_r may contain an invalid pointer if it refers to the result of a call that was never executed, e.g. a call made in a branch of an IF statement that was not taken.) Next, the offset is computed between the apparent type and the actual type, which can now be determined from the object. This offset is added to the partial offset, and the sum is added to the pointer before it is returned.

The mechanism for sharing the pointers to objects is as follows: When a call is parsed, the future for its result, f_v, is mapped to the type of the result in T. In addition, a new pointer object, p_r, is initialized that will point to the actual result. A mapping between f_v and p_r is also stored in T. When a later call is parsed that uses f_v as an argument, p_r is found by looking in T. A new Arg Node containing p_r is created for the Method Node representing that call. The Arg Node also contains the apparant type of the object so that the complete offset can be computed.

Because no mapping between variable futures and objects is stored in F, variable futures should be used only as arguments to other calls inside the control structure. A variable future will not refer to a valid object if it is used outside a batched control structure (as discussed in section 5.2.5). Any results that will be used as arguments to calls made after the end of the control structure should be stored in a cell object instead.

Figure 6-1 gives a schematic diagram of a Var Arg Node.

6.4.3 Method Nodes

A Method Node represents a single method call. The Method node contains: an Arg Node for the receiver of the call, an array of Arg Nodes for the arguments (if any), an Arg Node for the result (if any), and a pointer to the actual function to be called.

Figure 6-2 presents a schematic diagram of a typical Method Node.

Evaluation

When a Method Node is evaluated, it executes the method function and updates the result pointer. To evaluate a MethodNode the following steps must be performed:

- The Arg Nodes for the arguments are evaluated, generating an array of objects for the arguments.
- The Arg Node for the receiver is evaluated, and the actual function to use for the method call is determined by dispatching.
- The function is executed, and the result pointer is updated to point to the result.
- If the method signals an exception, the evaluation process is halted.

Method Node specializations

To improve the performance of Method Node evaluation, certain common cases were identified, and special Method Nodes are used in those cases.

First, if the receiver is constant (i.e. a handle, a basic value or a constant future), the method dispatch can be done at parse time. When this Method Node is evaluated, it does not need to recompute the function before calling it.

A second specialization is made if the method does not have a return value. In this case the evaluation need only check for an exception after the function has been called.

Finally, special Method Nodes are used if the method call has one argument, or if it has no arguments. If the single argument is constant, there is no need to use the Arg Node abstraction. Instead, the Method Node just contains a pointer to the object, which has already been offset properly.

For each of the specializations described above, there is a corresponding Method Node that does not contain that particular optimization, which I shall call its complement. A different method Node has been implemented for all the permutations of the specializations and their complements described above. Thus there is a special Method Node for a method call with a constant receiver that has one variable argument and no return values, etc.

6.4.4 While Nodes

A While Node represents a batched while loop. The Node contains: a list of Method Nodes for the condition expression, a pointer to the eventual result of the condition expression,

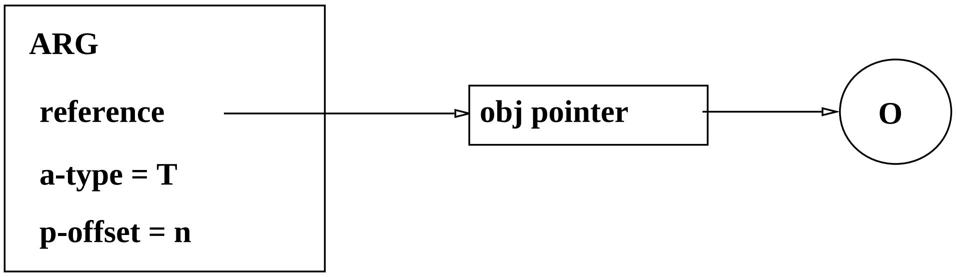

Figure 6-1: Var Arg Node

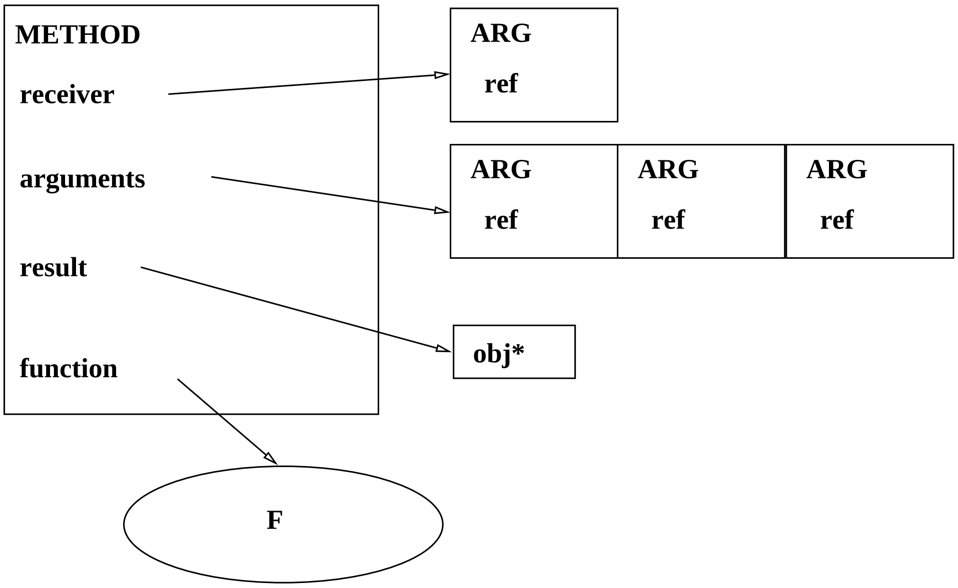

Figure 6-2: Method Node

and a list of Method Nodes for the body. The result of the last method call in the expression is the boolean condition of the loop.

Figure 6-3 presents a schematic diagram for a While Node.

Evaluation

A While Node is evaluated as follows: first, the expression is evaluated. If the condition is TRUE, the body is evaluated. This process is repeated until the condition is FALSE.

6.4.5 If Nodes

An If Node represents a batched if control structure. An If Node contains: a list of Method Nodes for the condition expression, a pointer to the eventual result of the condition expression, and lists of Method Nodes for the then-body, and the else-body.

Figure 6-4 presents a schematic diagram for an If Node. Note that the expression and condition are related in the exact same way as in the While Node.

Evaluation

First, the expression is evaluated. If the condition is TRUE, the then_body is evaluated; otherwise, the else_body is evaluated.

6.4.6 Foreach Nodes

A Foreach Node represents a batched for loop. A Foreach Node contains: a pointer to the iterator result, a single Method Node for the iterator, and a list of Method Nodes for the body of the loop.

Implementation of Iterators in Thor

In order to understand how a Foreach Node is evaluated, we must first understand the way iterators are implemented in Thor. This section presents a brief explanation of how an iterator is implemented in Thor.

A Thor iterator is implemented as a procedure that takes one extra argument (not listed in the signature of the iterator). This argument is a **closure** for the body of the for loop that is calling the iterator. The closure contains a function and an environment, which contains values for all the function's free variables, except one. The remaining free variable is the one through which the iterator will pass the value it yields on each iteration. After generating the value, the iterator calls the function in the closure with the value it has generated. When there are no more values to be yielded, the iterator returns.

For example, consider the following for loop in Theta:

```
int sum = 0

for i : int in 1.to(10) do
  sum = sum + i
```

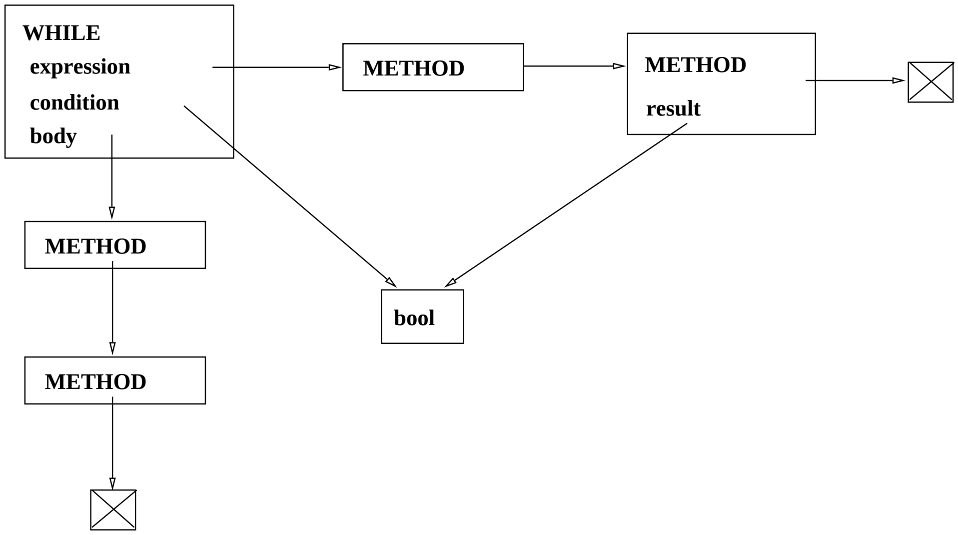

Figure 6-3: While Node

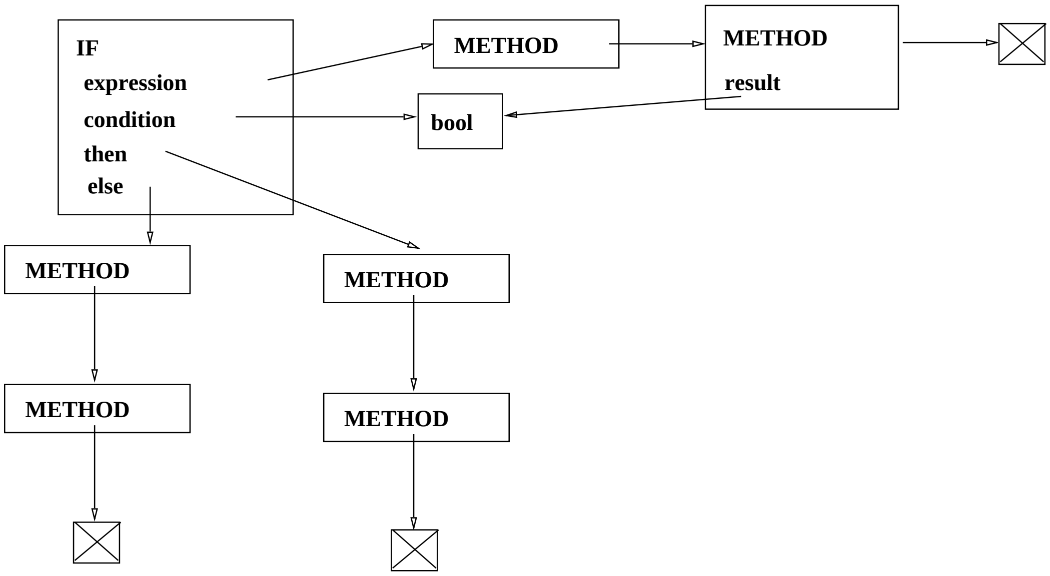

Figure 6-4: If Node

end

Figure 6-5 shows the iterator and closure in C corresponding to the example for loop above. Note that sum is passed by reference, so that a change in its value can be propagated to the next call to F. This code is generated by the Theta compiler.

Foreach evaluation

To use the mechanism described above for a Foreach Node, a closure must be constructed for the body of the Foreach Node, which can be passed to the iterator function. This is done as soon as the Foreach Node has been constructed. A closure, C, is created that will be used to evaluate the body of the Foreach. The function in C, F_c, takes 3 arguments: The first argument is the pointer (p_r) to the value yielded by the iterator. The second argument is the list of Method Nodes comprising the body of the loop. The last argument is the actual value (v) that is yielded by the iterator. When F_c is called, it sets p_r to point to v (or contain v if v is a basic value). This enables Method calls in the body (using Arg Nodes) to access v through p_r.

Figure 6-6 presents a schematic diagram of a Foreach Node.

A Foreach Node is evaluated by evaluating the Method Node for the iterator [1]. The function for the iterator is then called, with the closure for the body of the Foreach loop as its first argument. This function repeatedly generates a value, and calls the closure function F_c. When the iterator is done yielding values, it returns.

[1] In fact, as an optimization, the Foreach Node can be replaced with the Method Node for the iterator in the parse tree, eliminating an unnecessary level of indirection during evaluation.

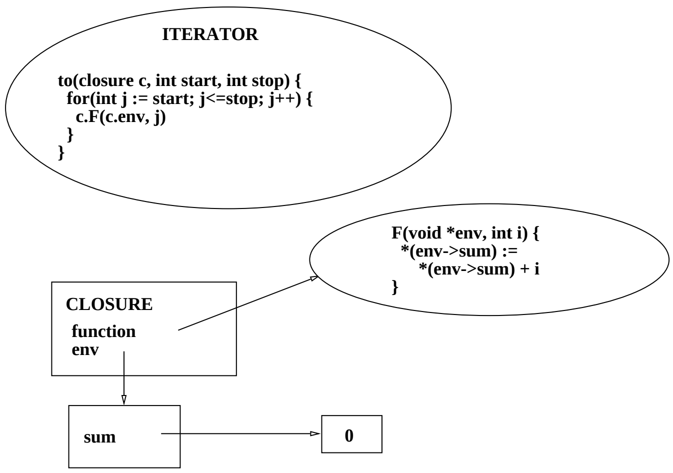

Figure 6-5: Iterator and closure

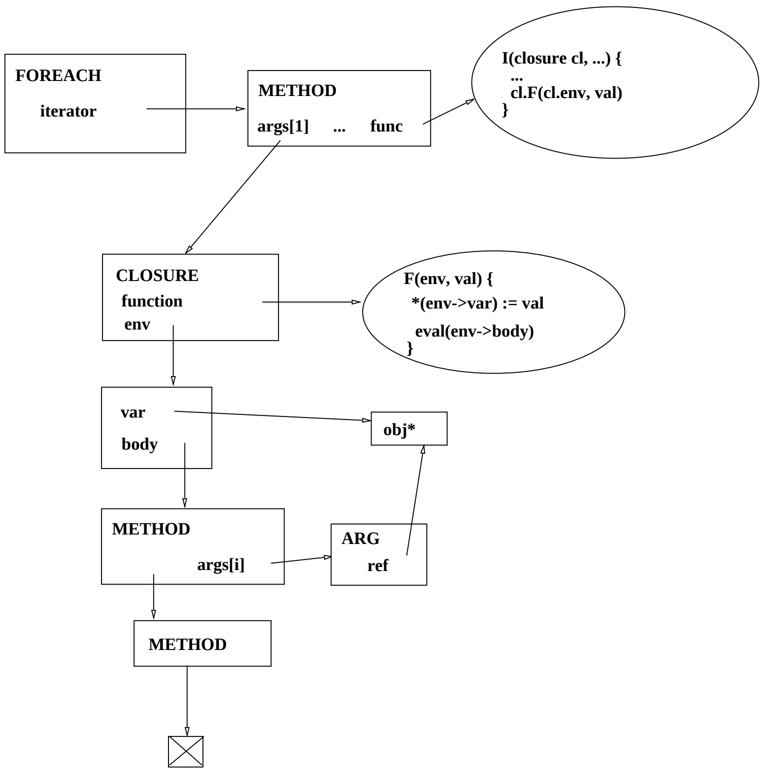

Figure 6-6: Foreach Node

Chapter 7

Performance Results

This chapter presents the results of measurements that demonstrate the efficiency of the BCS mechanism. The first set of experiments compares BCS to regular batched futures. The second set of experiments demonstrates the benefits of the Method Node optimizations described in Chapter 6.

7.1 Five ways to batch a loop

The first experiment shows the relative performance of BCS compared to batched futures and running the entire application inside Thor. Five different version of the same loop were evaluated:

1. A client side for loop
2. A client side for loop with perfect batching
3. A batched while loop
4. A batched for loop
5. A Theta version of the loop

For each loop, we measured the total time spent between the beginning and end of the loop. This was measured entirely on the client side, and thus includes some overhead of batching the calls on the client side. However, we expect this to be very small compared to the total time taken. All measurements were taken on a lightly loaded DEC Alpha AXP workstation. The optimizations described in chapter 6 were turned on for these measurements, and garbage collection at the Frontend was done only at start-up time .

Figures 7-1 and 7-2 show the total running time for each loop compared to the number of iterations executed for the loop. The actual measurements were taken at 1, 2, 4, 8, 16, etc. iterations. The error bars show a 95% confidence interval for each measurement.

Figure 7-1 shows the long term behavior of the loops. Figure 7-2 shows the behavior of the loops after only a few iterations. Note that the while and for loop curves are almost parallel to the inside-Thor curve. This means that the per iteration overhead is not much larger than than the optimum in both cases (Table 7.1). Also, note that both the while and for loop outperform the client loops after a very small number of iterations.

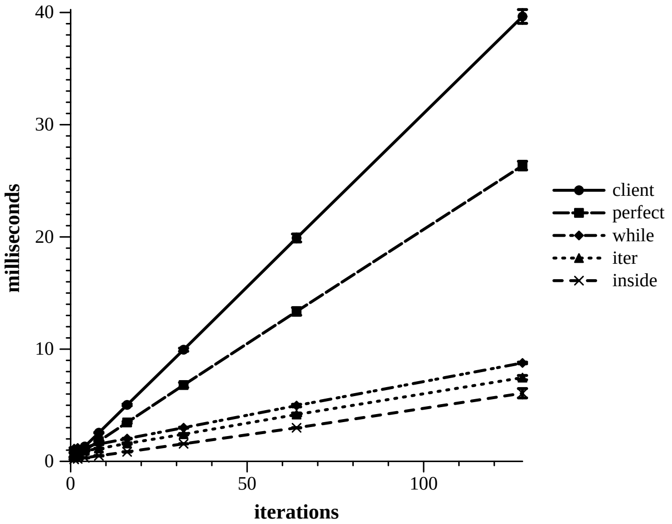

Figure 7-1: BCS Performance

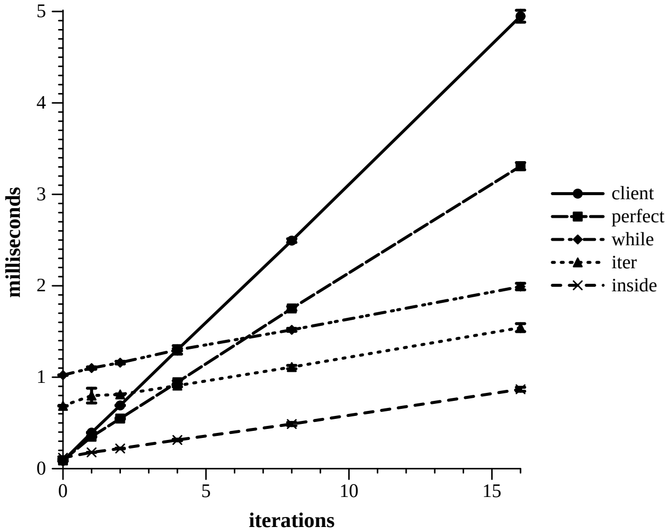

Figure 7-2: BCS Performance (enlargement)

Loop	Start-up (μs)	Cost/iter (μs)
client	85	308
perfect	90	200
while	1023	59
iter	687	51
inside	119	48

Table 7.1: Cost per iteration

Table 7.1 summarizes the start-up times, and the cost per iteration for the different cases. The start-up times are substantial for the BCS loops because the loops must be batched and interpreted by the Frontend, even though they end up not executing. From the data presented in Figure 7-1, the cost per iteration can be computed by taking the difference between the total time for two different iterations and dividing by the difference between the iterations. Note that the cost per iteration for both of the BCS loops is much less than for the client loops. The cost per iteration for the BCS loops is also not much higher than the cost per iteration for the Theta loop.

The remainder of this section explains each of the loops in more detail. Minor details in the code have been omitted for clarity. The first code fragment is a preamble for all the cases:

```
th_DataBase *data;
th_DataElem *res;

th_int *zero = new th_int(0);
th_int *one = new th_int(1);

for (int loop_iters = 1; loop_iters ≤ 128; loop_iters*2) {
```

The preamble declares some variables and creates promises to represent the values 0 and 1. The number of iterations is varied using the `loop_iters` variable.

In each case, the loop performs roughly the same amount of work. On each iteration, three calls are made to the `fetch` method on the same object, `data`. The `fetch` method performs a simple linear search on a "database" of users, looking for a string match with the argument `usr`. Because the `fetch` method does not return a basic value, all the calls are batched. In addition, each loop performs an addition on an integer, for reasons explained below. The while loop has to do some extra work to keep track of the number of iterations.

7.1.1 Client loop

This is a simple client for loop that sends a new batch to the Frontend on each iteration.

```
// Loop on the client side.
for (int i = 0; i < loop_iters; i++) {
  one→add(0);            // Force the batch across
  data→fetch(usr);
  data→fetch(usr);
  data→fetch(usr);
}
```

The first statement in the body of the loop, a gratuitous call to the integer add operation, causes the batch to be sent to the Frontend on each iteration, which is what would happen if the loop condition depended on a Thor call. A batched version of the same call is added to all the other versions of the loop. The main difference in cost between this loop and the next one (perfect batching) is the cost of a domain crossing per iteration; in addition, the veneer does a future remapping after each batch. It is clear from the previous figures that doing a domain crossing per iteration is very expensive.

In this case, the total cost for the loop is:

$$k(t_d + nt_c)$$

where k is the number of iterations, n is the number of statements in the loop, t_d is the cost of a domain crossing, and t_c is the average cost per call. The average cost per call is not an absolute measure, since it depends on the particular mix of calls; different calls do not necessarily have the same cost. However, it does give us some idea how the performance would be affected if we ran a similar loop with a larger number of similar calls in the body. Solving this equation for t_c, with $t_d = 70$ microseconds (measured) and $nt_c = 308$ microseconds (from Table 7.1), we get $t_c = 59.5$ microseconds. This includes the cost of doing a future remapping on each iteration, as well as the cost of batching and type checking. As we will see shortly, most of the cost of t_c is in this overhead.

7.1.2 Client loop with perfect batching

This is the same loop as the previous one, except that the call to `add` is batched.

```
// Loop on the client side.
for (int i = 0; i < loop_iters; i++) {
  one→Add(zero);             // Batch the call to add.
  data→fetch(usr);
  data→fetch(usr);
  data→fetch(usr);
}
```

In this version of the client side loop, there are no calls that return basic value results. Thus all the calls can be batched, causing the loop to be unrolled into the batch. The size of the batch is equal to the number of iterations the loop executes times the number of statements in the body. We refer to this case as "perfect batching". This case performs much better than the previous one, because the cost of the domain crossing is amortized over all the iterations of the loop, rather than across each iteration.

The total cost in this case is:

$$t_d + knt_c$$

In this, and all subsequent cases, t_d has a negligible impact on the total performance, since it is amortized across all the iterations of the loop. Again, t_c is computed as before. From Table 7.1, the cost per iteration, nt_c is about 200 microseconds. Thus, t_c is $200/4 = 50$ microseconds. Note that in this case, futures are remapped only once, namely after the loop has finished executing. Thus the cost of remapping is amortized much more effectively because it is more efficient to do one large remapping than many smaller ones. This accounts for most of the difference between t_c in this case, and t_c in the previous case.

7.1.3 Batched while loop

This is the version of the loop that uses the WHILE batched control structure.

```
// Batched while loop
th_cell<th_int> *sum = new th_cell<th_int>;

th_int *iters = new th_int(loop_iters);

sum→Put(zero);

WHILE(sum→Get()→Lt(iters))
  one→Add(zero);
  data→fetch(usr);
  data→fetch(usr);
  data→fetch(usr);
  sum→Put(sum→Get()→Add(one));
END_WHILE;
```

In this case, the entire batch consists of only the loop. Note the use of a cell, `sum`, to keep track of the number of iterations. Due to the use of the cell, the total number of calls per iteration, n, in this case (9) is greater than in either of the two client side loops (4). However, in this experiment, the while control structure still outperforms both client loops after a small number of iterations. This is due to the fact that the BCS mechanism amortizes some of the overhead of evaluating the calls across multiple iterations. In the batched futures versions, each call is parsed, type checked and evaluated individually. In the while control structure case the Frontend constructs a parse tree representation of the loop and evaluates it on each iteration. Because the parse tree has been highly optimized, the overhead of evaluating it on each iteration is much less than evaluating the individual calls using the batched futures mechanism.

The total cost of evaluating a batched while loop can be roughly modeled as follows:

$$t_d + nt_p + knt_e$$

where t_p is the cost of parsing and type checking each call in the loop, and t_e is the cost of evaluating each call. In this case, $n = 9$. t_p gives us some idea of the cost of parsing a call in BCS. Using the start-up cost for the while loop in Table 7.1, $t_d + nt_p = 1023$ microseconds. Thus, $t_p = (1023 - 70)/9 = 106$ microseconds. This includes the overhead of building a parse tree for the loop, etc.

The value for t_e gives us an idea what the cost of evaluating a call in BCS is. Using 59 microseconds for nt_e from Table 7.1, $t_e = 59/9 = 6.6$ microseconds. Note that this is quite a bit smaller than t_c in the previous two cases. Again, this is due to the fact that the parsing and type checking costs are not included in t_e, whereas they are in t_c. In addition, t_e is an average, and the calls to `Put`, `Get`, and `Lt` are very cheap compared to the cost of `fetch`.

7.1.4 Batched for loop

This is the version of the loop that uses the FOREACH batched control structure.

```
// Batched for loop
th_int *i;
```

```
FOREACH(i, one→To(iters))   // iters is same as for WHILE loop
  one→Add(zero);            // Batch the call to add.
  data→fetch(usr);
  data→fetch(usr);
  data→fetch(usr);
END_FOREACH;
```

A batched for loop is the ideal way to structure this particular loop. Whereas the while control structure required the use of a cell, introducing additional overhead, the for control structure allows a more concise description of the computation to be performed.

The performance of the for loop is modeled with the same formula as the while loop. In this case, $n = 5$ because of the extra call to the iterator `To`. In this case, $t_p = 123$, which is similar to t_p in the previous case. Again, the difference is due to the fact that the mix of calls is different, so the cost of parsing them is slightly different. The cost per iteration, 51 microseconds, is smaller than in the while loop because the number of calls is less. In this case, $t_e = 51/5 = 10.2$ microseconds. This is slightly higher than in the case of the while loop because in that case, the average included the calls to `Put`, `Get`, and `Lt`, which are much cheaper than the calls to `fetch`.

7.1.5 Theta loop

The final version of the loop is executed entirely on the Frontend side by making a single call at the client.

```
// Call Theta version of loop.
data→loop_fetch(usr, loop_iters);
```

This is the fastest possible implementation of this benchmark. The Theta code is given below. The `loop_fetch` method is defined on the Database type presented earlier.

```
loop_fetch(u : string, iters : int)
  for i : int in 1.to(iters) do
    1 + 0
    self.fetch(u)
    self.fetch(u)
    self.fetch(u)
  end
end loop_fetch
```

In this case, the cost of evaluating the loop is:

$$t_d + knt_x$$

where t_x represents the cost of executing each call. The cost per iteration is about 48 microseconds, and $n = 5$ again. Thus, $t_x = 48/5 = 9.6$ microseconds. Note that there is no future remapping in this case, as no futures are generated by the call to `loop_fetch`.

In this case the number of calls is the same as in the batched for loop. So, the difference between t_x and t_e from the previous case can be attributed almost entirely to the evaluation mechanism used in the batched for loop. Note that this cost is very small in this case (0.6 microseconds).

7.2 Benefits of optimizations

The experiment described in this section shows the performance gain due to the optimizations described in the previous chapter. In this experiment, a loop similar to the batched while loop described in the previous section was run for 100 iterations. The difference between this loop and the one in the previous section is that the number of calls to `fetch` is varied from 0 to 10, and there are no calls to `add`. However, the use of a cell to count the number of iterations remains.

Figure 7-3 shows the cost of evaluating and parsing the loop versus the number of calls to the `fetch` method. The curve labeled `not optimized` measures the expense of evaluation when Method Node (section 6.4.3) optimizations are not in effect. The graph labeled `optimized` shows the same experiment, but with Method Node optimizations turned on. In this case, the `fetch` method call translates into a Method Node with a constant receiver and a single constant argument. Calls to `cell` methods `get` and `put` have been optimized as well. The difference between the two curves at 0 calls to fetch is due to the fact that the calls to `Put`, `Get` and `Lt` are executed on each iteration.

Table 7.2 shows the average cost per call to the `fetch` method for the optimized and non-optimized schemes.

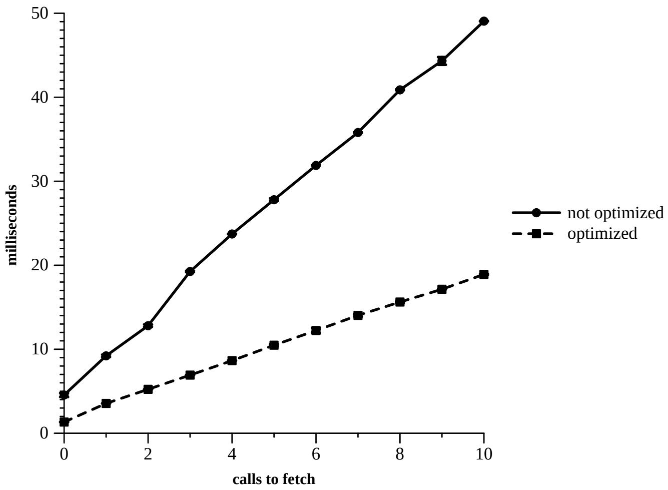

Figure 7-3: Method Node optimization

Scheme	Time (μs)
batched while	45
optimized while	17

Table 7.2: Cost per call to fetch

Chapter 8

Conclusions

The first part of this chapter presents a summary of the work done in this thesis, and some conclusions that can be drawn from it. The remainder of the chapter presents some suggestions for future work.

8.1 Summary

This thesis has presented two new mechanisms to increase the performance of cross-domain calls. The first, Basic Value Promises allows calls that return basic values to be batched, thus allowing for a larger batching factor, which implies better performance. Although in some cases promises can be used to increase the batching factor directly, we do not expect this to be their mayor use. Instead, promises are an integral part of the second mechanism presented in this thesis, Batched Control Structures (BCS). The BCS mechanism allows the client programmer to augment a batch with simple control flow information that is interpreted by the server. Using a batched control structure instead of a client language control structure can improve the batching factor by eliminating the need to communicate immediately with the server to obtain values critical to the control flow of the program.

Our performance studies have shown that both basic value promises and BCS can significantly improve the performance of a client program in Thor. Promises were used to increase the batching factor in the 2b traversal of the OO7 benchmark, thus increasing the performance. BCS was used to illustrate not only the possible increase in the batching factor, but also the reduced overhead associated with each call to Thor when using BCS.

This work is similar to Remote Evaluation (REV) [17] in that it allows a client program to send small "sub-programs" to be evaluated at the server. However, in Stamos' work, both client and server were written in the same programming language, CLU [11]. One of the goals of Thor is to allow clients written in different programming languages to share the same objects. In REV, the operations invoked by the client must either be exported by the server or provided by the shipped code itself. In Thor, the only operations that can be called by the client are those exported by Thor. BCS extends this calling mechanism by allowing control flow to be expressed in a batch of calls. However, BCS is by no means a general purpose programming language in the way that CLU is. Remote evaluation is different also in that its intended use is in distributing a computation among a number of servers to improve the performance of the application. In contrast, BCS is used in Thor to reduce the communication and evaluation overhead of remote calls; remote calls are the only way to interact with Thor.

Promises and BCS are points on a spectrum of client interaction with Thor. On one end of the spectrum is a client that uses Thor only as a repository of data objects. The client interacts with Thor by placing individual calls and waiting for their results. This is clearly an inefficient way of doing things. On the other extreme of the spectrum is an application that is written entirely in Theta, and stored in Thor as part of the database. A "client" in this case is nothing more than a small program that invokes a single "top-level" Thor procedure, which then does all the work. Presumably, this would be the most efficient way of working with Thor. However, this approach, has several drawbacks: First, it does not allow the client programmer to make use of the heterogeneity of Thor. The client programmer must learn Theta in order to write the application. We envision that most application programmers will want to use the language they are already familiar with. Second, writing all or part of the application in Theta requires that it be stored persistently and named inside Thor so that it can be used multiple times. Instead, by providing the programmer with tools such as batched futures and BCS, an application can be written succinctly and efficiently in the client language against a set of predefined types stored inside Thor.

8.2 Future Work

The next three sections describe some possible extensions to the BCS mechanism.

8.2.1 Batched break statement

One very useful extension to the current BCS mechanism would be the addition of a **break** statement that terminates the execution of the nearest enclosing loop. The effect of break can always be achieved in a while loop by encoding the condition for when to break into the loop condition itself. However, in the case of a for loop this is not possible. Thus, break would be the only way to achieve early termination in the case of a for loop. This is important when, for example, an iterator is used to do a search, in which case we want to terminate the search as soon as a match is found.

At the veneer, the break statement would be implemented as a procedure that places a BREAK marker in the batch. At the Frontend, the parse tree would contain a Break Node to represent the break statement. When evaluated, the Break Node sets flag. This flag is checked by each evaluation function before evaluating its corresponding Node. The evaluation functions for If and Method Nodes return immediately if the flag is set, without evaluating their respective nodes. The evaluation function for the While Node checks the flag on each iteration. If the flag is set, it clears the flag, then returns immediately. Thus, evaluation continues with the Node following the While Node in the parse tree. In the case of Foreach, the iterator checks the flag on each iteration. If the flag is set, it clears the flag and returns.

8.2.2 Batched continue statement

The semantics of continue are that execution continues with the beginning of the next iteration of the loop. Continue would be implemented in a similar way to Break. The veneers provides a CONTINUE statement, which can be called from within a batched

While or Foreach loop. At the Frontend, a continue flag is used. If the Continue Node is evaluated, it sets the flag. Subsequent calls to If and Method Node evaluation functions check the flag and return immediately if it is set. The evaluation functions for the While and Foreach Nodes clear the flag after each iteration.

8.2.3 Batched Procedures

One inconvenience of BCS is that there are no batched procedures. If a sequence of code must be performed multiple times, the code has to either be duplicated inside the BCS, or has to be rewritten as a Theta procedure that can be called directly by the client. Although it is possible to simulate recursion in BCS, it can be quite cumbersome.

It would not be too hard to extend the current BCS mechanism to include procedures. The basic idea is to create a parse tree for a batched procedure, and associate it with a name provided by the client. The client program can cause the parse tree to be evaluated at different times by invoking it explicitly, using the name.

To illustrate this mechanism, we discuss the case of a procedure with one argument and with no results. The real mechanism would provide support for multiple arguments and at least one result. A new PROCEDURE construct would be used, which takes three arguments: a name for the procedure, a single argument to the procedure, and a name for the type of the argument. The argument can be of any type. The name for the argument type is used by the Frontend to look up the corresponding type. This type is stored in the parse tree Node for the procedure, so that the type checking algorithm can verify that the argument passed by the client is of the right type.

For example, here is how a batched procedure would be created for the swap procedure mentioned in chapter 4:

```
th_cell<int> temp = new th_cell<th_int>;

PROCEDURE(``swap'', p, ``point'')
  temp->Put(p->X());
  p->Set_x(p->Y());
  p->Set_y(temp->Get());
END_PROCEDURE
```

To call the batched procedure, the client calls the `invoke_procedure` function provided by the veneer:

```
  th_point p = ...;

  invoke_procedure(``swap'', p);
```

8.3 Comparrison with Sandboxing

An alternative to providing safety without running in separate protection domains is presented in [15]. This mechanism, software-based fault isolation, popularly called "sandboxing," inserts software bounds into unsafe client code to guarantee that the client does not access memory outside its designated area. It reduces the per call overhead by eliminating the need for a domain crossing on each call. Using sandboxing in Thor instead of batching would not obviate the need for type checking, but it would eliminate the overhead of constructing and parsing a batch of calls. On the other hand, the BCS mechanism allows the server to construct a highly efficient representation of the computation that is repeatedly evaluated. So, it is not clear whether sandboxing would produce larger performance benefits than BCS, or even batched futures. A thorough comparison of the two schemes would allow us to answer that question, and help decide whether the drawbacks of sandboxing are worth the gains.

Bibliography

[1] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[2] Andrew P. Black and Norman Hutchinson. Typechecking polymorphism in Emerald. Technical Report 91/1, Digital Equipment Corporation, Cambridge Research Laboratory, December 1990.

[3] P. Bogle. Reducing cross-domain call overhead using batched futures. Master's thesis, Massachusetts Institute of Technology, 1994.

[4] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Also in *Readings in Object-Oriented Database Systems,* S. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.

[5] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, DC, May 1993.

[6] M. Castro. Object invocation performance in thor. Programming Methodology Group Memo to appear, Laboratory for Computer Science, MIT, Cambridge, MA, 1995.

[7] Mark Day. *Client Cache Management in a Distributed Object Database.* PhD thesis, Massachusetts Institute of Technology, 1995. Forthcoming.

[8] Mark Day et al. *Theta Reference Manual.* Programming Methodology Group Laboratory for Computer Science, MIT, Cambridge, MA, 1994. **To appear.**

[9] J. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18:683–696, December 1975.

[10] B. Helfinstine. A Modula–3 veneer for Thor, 1994.

[11] B. Liskov et al. CLU reference manual. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, volume 114. Springer-Verlag, Berlin, 1981.

[12] B. Liskov, R. Gruber, P. Johnson, and L. Shrira. A highly available object repository for use in a heterogeneous distributed system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems Design, Implementation and Use*, pages 255–266, Martha's Vineyard, MA, September 1990.

[13] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Languages Design and Implementation.* ACM, June 1988.

[14] Andrew C. Myers. Fast object operations in a persistant programming system. Master's thesis, Massachusetts Institute of Technology, January 1994.

[15] T. Anderson R. Wahbe, S. Lucco and S. Graham. Efficient software-based fault isolation. *Proc. Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, December 1993.

[16] C. Schaffert et al. An introduction to Trellis/Owl. In *Proceedings of the ACM Conference on Object Oriented Systems, Languages and Applications*, Portland, OR, September 1986.

[17] J.W. Stamos and D.K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.