Reducing Cross Domain Call Overhead Using Batched Futures

Phillip Bogle and Barbara Liskov *
(pbogle, liskov)@lcs.mit.edu
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

In many systems such as operating systems and databases it is important to run client code in a separate protection domain so that it cannot interfere with correct operation of the system. Clients communicate with the server by making cross domain calls, but these are expensive, often costing substantially more than running the call itself. This paper describes a new mechanism called batched futures that transparently batches possibly interrelated client calls. Batching makes domain crossings happen less often, thus substantially reducing the cost. We describe how the mechanism is implemented for the Thor object-oriented database system, and presents performance results showing the benefit of the mechanism on various benchmarks.

1 Introduction

An important issue in the design of software systems is preventing untrusted clients from interfering with the correct operation of servers. Systems ranging from databases to operating systems solve this problem by requiring that clients run in their own protection domains (typically a separate process) and communicate with servers via cross-domain calls. An ill-behaved client is thus prevented from corrupting data structures, reading private information, or otherwise interfering with the correct operation of servers. Protection domains thus increase the modularity, security, and debuggability of the system.

However, crossing protection domains is expensive; existing interprocess call implementations are at least an order of magnitude slower than direct calls [11, 2]. This is prohibitively expensive for lightweight operations; much more time is spent crossing protection domains than getting work done. The problem is likely to get worse with advances in software and hardware technology: context-dependent optimizations in hardware such as pipelining and caching make context switches all the more expensive, while increasingly complex software systems make them all the more important.

This paper presents a general mechanism, called *batched futures*, for reducing the cost of cross-domain calls. The basic idea is that certain calls are not performed at the point the client requests them, but are instead deferred until the client actually needs the value of a result. By that time a number of deferred calls have accumulated and the calls are sent all at once, in a "batch". In this way we can turn N domain crossings into one, and user code runs faster as a result. Our mechanism makes the batching transparent to client applications and allows later calls to make use of the results of earlier calls. In addition, it can be used when the client and server run on the same machine or different machines.

Batched futures are particularly useful for interacting with object-oriented databases. An object-oriented database can potentially provide much safer sharing than is possible in a file system or conventional database because it can know about objects' types, and objects

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136, and in part by the National Science Foundation under grant CCR-8822158.

OOPSLA 94- 10/94 Portland, Oregon USA
© 1994 ACM 0-89791-688-3/94/0010..$3.50

can be encapsulated, and accessed only by method calls. As a result, objects entrusted to the database are less likely to be corrupted by the client applications that share them, *provided* applications cannot violate encapsulation. Encapsulation can be preserved by running the application in a separate domain, so that it can interact with objects in the database only by means of cross-domain calls to invoke their methods. This approach is likely to be expensive, however: methods calls often perform very little work (for example, a call might just look up the value of an instance variable), so that the time to execute calls will be dominated by the domain-crossing overhead. Deferred calls therefore offer the potential to significantly improve performance.

We describe batched futures in the remainder of this paper. We begin in Section 2 by discussing related work. Section 3 describes Thor, an object-oriented database system that served as the context for our work. Section 4 describes futures; Section 5 describes how they are implemented. Section 6 shows the speed-ups obtained by futures by presenting the results of experiments on various benchmarks; the results show that the mechanism obtains almost a two-fold speedup in an unfavorable case and can do considerably better. Section 7 discusses extensions to our mechanism, including a way to defer more calls. We conclude in Section 8 with a discussion of future research directions.

2 Related Work

The batched future mechanism is based on two earlier lines of research. The first is the future mechanism of the parallel programming language Multilisp [9]. In Multilisp, the construct *(future E)* forks a parallel thread to evaluate the expression E and immediately returns a future to the main program as a placeholder for the eventual value of the expression. The future is overwritten with the value of E when its evaluation is complete. Batched futures are not primarily aimed at concurrency (although as we shall see they can allow concurrency) but instead delay calls and make a batch of them at once, hence the name. (In other words, batched futures do lazy rather than eager evaluation.)

The other predecessor is work on communication streams, such as pipes [8] and Mercury streams [14], in which remote calls are collected and then sent over in a batch in order to reduce delay to the caller. This work is similar in spirit to ours, but is concerned with masking network delay, and requires explicit user control. A future-like mechanism called promises is proposed in [15] as a way to control streaming in client code. Stream and promises are more limited than our mechanism because they do not allow later calls to refer to the results of earlier ones.

An invocation chaining mechanism similar to batched futures is proposed in [1], but the paper only provides a sketch of the mechanism and many details, e.g., of storage management (how and when to reclaim the storage used by the mechanism so that it does not grow without limit), efficiency, transparency, and type safety, are not worked out.

The software based fault-isolation mechanism described in [17] allows an untrusted module to run safely in the same address space as protected data and code by putting guards around each jump, load, and store in the untrusted code to prevent it from accessing or jumping to memory outside of its own address segment. The entire client program is slowed down as a result of the restrictions, but the performance penalty is relatively low. When the isolated code calls into the protected code, it does so only to defined entry points using a special mechanism; the cost of such calls is also low. This approach will probably out-perform ours in many cases, but it is more difficult to use since it requires either modifications to the compiler of the client language or modifications to all the client binaries. Also, our approach can be used even when the client code runs on a different machine than the server.

3 Environment

This work was done in the context of Thor [13]. Thor is an object-oriented database system that aims to satisfy two sometimes conflicting design goals: it must provide strong guarantees about the security and integrity of its objects while remaining cheap enough to use even for lightweight objects and operations. Thor objects are implemented in an object-oriented programming language called Theta [7]. Applications that use Thor objects can be written in arbitrary programming languages, and in

fact a single application can have components implemented in different programming languages and yet sharing Thor objects. Thus Thor provides for multilingual sharing of its objects.

Thor objects are encapsulated and can be accessed only by method calls. Each object has a type that determines its methods; Theta provides a set of built-in types and users can define new abstract types. A description of the interface of each type is stored in Thor. Method calls happen within transactions; the application indicates when to commit (or abort) the current transaction and start a new one. Objects exist in a persistent universe. The universe has a persistent root; all objects reachable from the root are stored in highly reliable and highly available storage and the storage of unreachable objects is reclaimed automatically by the garbage collector.

Method calls return either handles or basic values. Basic values are immutable, unstructured data, such as integers, that are copied into client space and used there directly. For most objects, however, the client receives only an opaque pointer, known as a *handle*, that identifies the object and can be used to invoke its methods. A handle is actually an integer index into a table H in the database that maps handles to actual objects. The first time an object is returned as a result of a client call, the database chooses a handle for it by selecting a free slot in H. When handle j is later used in a call, the database looks in H[j] to find the corresponding object. Handles are valid only for the duration of a client session, so that H does not need to be persistent.

When the client begins a new session, the *find_wellknown* call enables it to obtain handles for a few well-known objects by name, most importantly the root of the persistent universe. In general, however, the client finds objects by *navigation*: a method call follows a pointer from an object and returns a handle to some other object as a result. For example, the root object is a directory that can be navigated using directory methods (e.g., the lookup method). Typically an application will do many stages of navigation before reaching an object where the values are of interest.

Thor is actually implemented using a client/server structure, in which persistent objects are stored at server machines. Thor runs a process at the client machine that maintains a cache containing copies of Thor objects. The Thor process runs method calls made by the application (which runs in a separate process); the method calls read and modify the cached copies. The Thor process fetches copies of objects into the cache when there is a cache miss, and prefetches objects into the cache in expectation of future use [6]. Modified copies of cached objects are copied back to the servers when transactions commit.

3.1 Veneers

Languages that interact with Thor are augmented by a thin layer that we call a *veneer*. We have implemented veneers for C, C++, Lisp, Perl, and TCL. Here we sketch how veneers work; more details about veneers and how they are implemented can be found in [4].

A veneer provides a small set of built-in commands (for example, there is a command to commit a transaction) plus mechanisms for referring to Thor objects and calling Thor methods. The veneer is nothing more than a set of client routines and types, and does not require a preprocessor or modifications to the compiler for the client language. It is produced by a *veneer generator*, which is similar to the stub generators used in RPC (remote procedure call) systems [3]. A veneer generator for a particular client language maps Thor objects and operations to *stub objects* and *stub functions* in that language. When a client program calls a stub function, the function makes a cross-domain call to invoke the corresponding operation in the database.

Handles are encapsulated inside of stub objects, which are allocated in the heap. Application programmers are supposed to follow certain conventions when using stub objects, although no harm comes to Thor if a client violates these conventions because of the explicit checking of calls that is discussed below. Stub objects are not intended to be manipulated directly by clients. Furthermore, clients are supposed to only copy pointers to stub objects and not the stub objects themselves, and, in a language with explicit deallocation, they should call the special *free_object* operation, provided by the veneer, to deallocate a stub object.

The veneer maintains its own handle table, VH,

```
int nth(th_list* l, int n)
{
  while (--n)
      l = l->next();
  return l->first();
}
```

Figure 1: Client code for the *nth* function.

which maps handles to stub objects. When a call to Thor returns a handle, if the handle is already defined in VH, the veneer returns the previously created stub object that contains it; otherwise the veneer creates a new stub object containing the handle, stores a pointer to the stub object in the handle's slot in VH, and returns the new stub object. This implementation ensures that there is one stub objects per handle in the veneer (assuming the client does not copy stub objects).

Handles are treated differently in different veneers. In a language without compile-time type checking (such as Lisp), all handles are treated alike, and no type checking is performed. In a language with a static type system (such as C++), handles are encapsulated in stub objects with distinct types mirroring those of the Theta type hierarchy, allowing compile-time type checking of Thor calls in the client program. For example, for a Thor list of integers with methods *first* and *next*, the C++ veneer will provide a corresponding C++ class *th_list* and two associated stub methods, also named *first* and *next*. C++ code to return the *nth* item (an integer) of a *th_list* is shown in Figure 1.

To ensure safety, Thor does runtime checking of each client call coming from an unsafe language (whether it has strong type-checking or not) to be sure handles are legitimate, objects have the methods being called, and the caller has supplied the proper number and types of arguments. (An unsafe language is any one in which object encapsulation is not enforced, e.g., because runtime type checking errors are possible, or because pointer manipulations can undermine the type system.)

4 Batched Futures

A future can be viewed as a reference to the eventual result of a call. Like the actual result, it can be passed as an argument to other calls, included in data structures, and so forth. If the actual value referred to by the future is required by the client and is not yet computed, the client waits until it is available. As mentioned, futures were introduced in the parallel programming language Multilisp to allow caller-callee parallelism, but we use them for a different reason: to defer calls so that they can be batched.

An important point to realize is that for many calls, the actual return value is not of immediate interest to the client. This is especially true in an object-oriented system with encapsulation: The particular value of a reference to an encapsulated object is *never* of any interest, since the reference can only be used as an argument or recipient of another call.

The basic idea of our batched futures design, then, is to batch calls as long as they return handles. When the client makes such a call, the stub function records information about the call, and returns a *future* to the client instead. Later calls in a batch can refer to results of earlier ones using futures and thus a sequence of interrelated calls can be batched together. As soon as the client makes a call that will return a basic value, or commits a transaction, the veneer sends the entire batch of calls to the database in a single domain crossing. As the database processes each call, it makes a mapping between the result and the corresponding future to allow the result to be retrieved for later uses of the future.

For example, the *nth* function in Figure 1 would normally require $n + 1$ cross-domain calls (each of which requiring two domain crossings). With batched futures, the same code requires only a single cross-domain call. Inside the loop, the client code makes *no* domain crossings; the *next* stub function simply returns futures $f_1 \ldots f_n$ to stand for the results of the calls and adds information about each call to a queue of batched calls. Each future is used as the receiver (i.e., first argument) of the following call. Finally, the client calls *first*, which returns a basic value. The stub function for *first* sends the batch of calls to the database, which evaluates the set of batched calls and sends back the requested result for *first*. This entire process is depicted in Figure 2.

In essence, the veneer has constructed a simple program that recreates the effects of a set of possibly interrelated calls when evaluated in the database. The

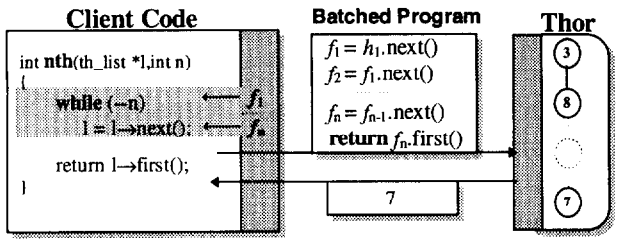

Figure 2: Batching interrelated calls using futures.

program has just a few simple actions:

- calling an operation;
- assigning the result of an operation to a future usable as an argument to later calls;
- returning the result to the client .

The database's job is to interpret this program efficiently.

This example may not be realistic, since the *nth* function might be included as an operation in the database, but it illustrates an important point: the function navigates a chain of pointers before reaching a point where actual values are of interest. Clients often navigate other, less predictable chains of pointers. It is not realistic to expect a built-in database operation for every chain of pointers that a client might follow, both because it is difficult to anticipate every useful chain and because to do so would create a very cluttered interface. Batched futures help resolve the dilemma: type interfaces can consist of a logical set of operations, possibly fine-grained, that are combined based on the particular needs of the application.

Note that the mechanism depends upon knowing the signatures of the database methods, so that the veneer can tell whether to send a call immediately or defer it. This information is available in the type interfaces stored in Thor and embedded in the stub functions. Stubs that return handles defer their calls for later execution; other stubs cause the execution of their call and the preceding batch of deferred calls.

Batching calls allows the server to "see the future" by looking ahead in the current batch of calls. This might enable it to improve performance. For example, a file system could reorder read requests to optimize disk head motion or a three-dimensionsal rendering system could avoid performing an expensive rendering if it determined that a later call would obscure it. Thus batching can have other benefits in addition to reducing the number of domain crossings.

5 Implementation of Batched Futures

There are three key issues in the implementation of batched futures:

1. How to represent futures in the veneer. Futures must be represented in such that a way that they are interchangeable with handles in stub objects.
2. How to maintain the mapping between futures and actual objects in the database so that the database can associate a future used as an argument in a call with the corresponding result.
3. How to limit the size of the mapping, since each call returns a new future and the mapping could potentially grow without bound.

We consider each of these issues in turn.

5.1 Representing Futures

A future is just an integer chosen by the veneer to stand for the eventual result of a call. To distinguish futures from handles, we tag them using the sign bit: futures are negative, handles positive.

Like handles, futures are encapsulated in stub objects, to which the client program is given pointers. Because of this level of indirection, there is only one instance of each future; on assignment, clients copy pointers, not the stub objects themselves, just as with handles.

5.2 Mapping Futures to Objects

Two parallel tables maintain the mapping between futures and objects, analogous to the H and VH tables that map handles to objects and stub objects. The veneer future table VF maps futures to stub objects, and the database future table F maps futures to actual objects.

Slots in VF are initialized when a stub function returns a future to the client. Such a stub function:

1. Increments a global counter to determine the index i of the future.

2. Batches a message describing the call and the choice of future i to hold its result. (The database will receive the message as part of the next batch of calls.)

3. Allocates a stub object o that contains future i.

4. Stores a reference to o in VF[i].

5. Returns o to the client program.

The database initializes slots in F when it processes a batch of calls. Each deferred call specifies the future index i that should be mapped to the result; after the database processes the call, it stores the resulting object in F[i]. When future i is used as an argument to a call, the database looks in F[i] to find the corresponding object. Because the database processes the calls in the same order they were made, the arguments of calls are determined by the time the database processes them.

The following is a more detailed description of the steps performed by the database in processing a call. The database:

1. Reads the method name and arguments.

2. Looks up each (non-basic) argument in H or F, depending on whether the index is positive or negative.

3. Type checks and performs the call.

4. If the call is the last of a batch, sends the result back to the client. Otherwise stores a reference to the call result in F[i], where i is the index specified by the client.

Note that the only overhead that has been added to the database is a couple of inexpensive conditionals. This overhead is dwarfed by the amount of time saved in avoiding a domain crossing.

5.3 Limiting the Size of the Future Mapping

Because each deferred call returns a new future and a future could be used as an argument to any later call, tables F and VF can grow arbitrarily large. Our implementation solves this problem by periodically replacing futures with the corresponding handles. After the number of futures passes a limit, the veneer flushes the pending invocations and piggybacks a request to the database to send the object handle equivalents for all futures currently in use. The veneer uses the pointers in VF to update the stub objects; because the client copies pointer to the stub objects, and not the stub object themselves, we do not have to worry about updating and tracking multiple copies of each stub object. The veneer and the database can then safely reclaim all slots in F and VF. Thus futures require only a constant amount of additional space relative to normal calls. The database could actually send back the corresponding handles after every batch of calls. However, there are performance advantages to sending handles in larger batches. It is cheaper to do one big read than a lot of small ones. Also, the longer the database waits, the greater the likelihood that the client has freed futures in the batch (in a language with explicit deallocation), allowing the database to avoid sending back the corresponding handles, as discussed below.

5.4 Stub Object Storage Management

Managing storage associated with stub objects containing futures is not a problem if the client language is garbage collected. The stub object will not be discarded while still referenced from VF, even if the client program no longer refers to it, but will be subject to garbage collection as soon as it is updated with the correct handle and the slot in VF is cleared.

For languages with explicit deallocation, the *free_object* operation provided by the veneer clears the entry in VF as well as reclaiming the object storage. When the veneer replaces futures with handles, it skips the entries in VF that have been cleared. It can also tell the database which futures have non-empty entries in VF, saving the expense of sending back handles for futures that have been freed.

There is one other concern regarding stub objects containing futures. When stub objects contain handles, the veneer ensures that there is only one stub object for each database object. The same property *does not* hold

for stub objects containing futures. This is a necessary consequence of the fact that the veneer does not know what the actual handle will be at the time the stub object is created. Therefore multiple stub objects can refer to the same database object. Such redundant stub objects are a problem only if the client has a very large number of *active* references that correspond to a small number of actual objects. (Inactive references are not a problem because in that case the stub objects will be freed.) Presumably, this will be not be a common occurrence. In general, we expect that most futures are only temporaries (that is, intermediate navigation pointers) and users won't want to hold onto them.

If redundant stub objects are a problem, the veneer can support an operation that takes a reference to a stub object, frees the stub object if it is redundant, and returns a pointer to the corresponding 'canonical' stub object for that object's handle. (The operation requires, of course, that the client has no other references to the redundant stub object.) Alternatively, a customized garbage collector can do the same thing safely and automatically by redirecting pointers to redundant objects to the canonical one so that redundant objects can be freed.

5.5 Shared Memory Optimizations

Batched futures can be implemented using Unix pipes. In this case the veneer just writes each message to the stream but does not flush the stream until it needs a result. However, it is faster to use shared memory if it is available. Each deferred call is recorded in a shared memory buffer, and data written by one side is always immediately visible to the other side without any need for flushes. (To obtain optimal performance, it is important that when one process blocks waiting for a result, the other is awakened quickly. Our implementation uses shared-memory semaphores to obtain this effect.) With this implementation, the database can begin working on deferred calls whenever it has time, even if the veneer is not yet waiting for the result of the last call in a batch. For example, on a multiprocessor this approach allows the database to process calls in parallel with the veneer.

Another shared memory optimization permits us to get rid of the F and VF tables and future remapping entirely. The tables exist only to allow futures used in later calls to be mapped to the results of earlier calls. The same effect can be achieved by allocating stub objects in shared memory. Rather than passing a handle or future, the veneer passes a pointer to the stub object, which the database dereferences. When the result of a batched call becomes available, the database stores the handle in the stub object allocated for the result, effectively performing the future remapping step for that object immediately.

The use of shared memory might raise the concern that the security of the system has somehow been compromised. Note, however, that the worst a client can do is overwrite a stub object in shared memory, which it could do just as easily when the stub object was in its own private address space. The database is not exposing any more information than it was before, and its runtime type checking will continue to prevent the client from making illegal calls.

6 Experimental Results

In this section we characterize the gains that can be expected from using futures, and present experiments showing the benefit obtained on various workloads. We give a simple model of the system's performance, provide results that show the predicted performance across a range of batching factors and domain crossing costs, and finally give results based on a standard benchmark.

The experiments were compiled using DEC C++ and run on a lightly loaded Alpha AXP3000 running OSF/1.3. The Thor experiments used a warm object cache because we wanted to measure just the amortized cross-domain call time, and not additional delays to fetch objects.

6.1 Performance model

The average cost of a call can be modeled by the formula

$$t = t_c + t_d/B$$

where t_c is the cost of running a call, t_d is the cost of the a pair of domain crossings, and B is the "batching factor," the total number of calls divided by the number of pairs of domain crossings. When there are no futures, B =

1 (since each call requires a pair of domain crossings); as we defer calls, B increases and the average cost of a call goes down. Note that the model assumes that t_d is independent of B. In other words, the amount of time it takes to switch between domains is independent of the amount of data (that is, the number of batched calls) being transferred between the two domains. This assumption is approximately true in actual systems; for example, in the case of a remote call system, an entire batch can be sent in one message up to some size limit beyond which more than one message will be needed.

The model predicts that, as B increases, the average cost of a call will asymptotically approach t_c, dropping rapidly at first and then with increasing slowness as t_d/B goes to zero and t_c begins to dominate the total cost of the call. The key points to note are that t_c provides a lower bound on the average cost per call, and that the larger the ratio of t_d to t_c, the more a system has to gain from batched futures, but also the higher the batching factors it must achieve before the domain crossing overhead t_d/B becomes negligible. For example, if the ratio of t_d to t_c is r, then a batching factor of r will yield performance that is within a factor of two of the optimal value.

6.2 Potential Performance Gains

The best case for batched futures is a client program in which all of the cross-domain calls can be batched. To experiment with the potential performance improvements that might be achieved using futures, we considered the *nth* function described earlier. Each operation in the *nth* function returns a handle, so that all of the calls can be batched, leading to an arbitrarily large batching factor B.

To show how the speedup provided by batched futures varies across a range of values for t_c and t_d, and also to demonstrate a range of systems to which batched futures are applicable, we considered three systems:

1. **Local IPC:** A simple client-server system with a very low t_c value and a comparatively high value for t_d. The client and server run in different process on the same machine and communicate using a shared memory buffer. The server, written in C++, implemented just the essential elements necessary to run the experiment: a linked list, a very simple dispatcher, a handle table, and a future table. It did not implement type checking, garbage collection, concurrency control, persistence, or any other features of Thor as a database. The stub functions and the client program were essentially the same as those in the case of Thor, however.

2. **Local Thor:** Thor running in its typical configuration, in which the application process and interacts with a separate Thor process at the client machine. The Thor process caches copies of persistent objects and performs client calls. The t_c for this system is significantly higher than that of the first system primarily because type checking and dispatching of calls in the current prototype is slow and needs to be optimized.

3. **Remote Thor:** Here there is no Thor process at the client machine, so that every application call requires a network communication. We may run Thor in this fashion if the client machine has a memory too small to be an effective cache, or does not have sufficiently safe protection domains.

The possible benefits of batched futures depend on the values of t_c and t_d in the different systems. These values are summarized in the Figure 3, which gives approximate values for t_c and t_d in microseconds for each system. The values for t_d were estimated by observing the difference between average call time for B=1 and B=2, which (using our formula) works out to be t_d/2. ("Average call time" was determined by taking the actual *elapsed* time for the list descent divided by the number of calls, averaged over 100 trials. The variation from trial to trial was small as long as the load on the CPU was low.) The value for t_c was then estimated by subtracting t_d from the average call time for B=1.

The observed performance of the systems is shown in Figures 4, 5, and 6. As predicted by the model, the average time per call drops rapidly at first and then approaches t_c for that system with increasing slowness; the shape for each graph resembles the graph of 1/B scaled by t_d and shifted up by t_c.

In the local IPC system (Figure 4), batched futures lead to a greater than tenfold increase in performance

	t_c (μ sec)	t_d (μ sec)
Local IPC	10	126
Local Thor	75	130
Remote Thor	100	905

Figure 3: Approximate values of t_c and t_d for three systems.

for sufficiently large B. The maximum speedup for local Thor (Figure 5), is around 3. The speedup is small because operation marshaling, type checking, and dispatching in Thor are currently expensive, leading to a high value for t_c even though the operations themselves are simple. When the operation dispatcher is optimized, we can expect to see speedups from batched futures closer to those seen in the simple IPC system.

Even Thor's relatively large value of t_c is dwarfed in comparison with the cost of a network communication (Figure 6). In that case, batched futures can again provide to a tenfold speedup. If batched futures were used in a system that had the t_c of our IPC system and the t_d of a system communicating over a network, we would expect to see up to a 90-fold performance improvement. Obtaining this speedup, however, would require hundreds of operations being combined in each batch.

6.3 A Less Favorable Case

Clearly, not all applications are as favorable for batched futures as list descent. For example, in a graph traversal the number of nodes connected to the current node may need to be know immediately, meaning that many calls cannot be deferred and the batching factor is necessarily low. In this section, we give an example of a less favorable application to show the limitations of the batched futures mechanism as currently defined and to suggest ways it can be improved.

As a representative application, we implemented various traversals from the OO7 suite of benchmarks for object-oriented databases [5]. The OO7 database consists of a set of interconnected parts, arranged in a hierarchy of complex assemblies, base assemblies, composite parts, and finally atomic parts. The operations on the parts and assemblies are quite simple: they either return a connected part, or a scalar attribute, such

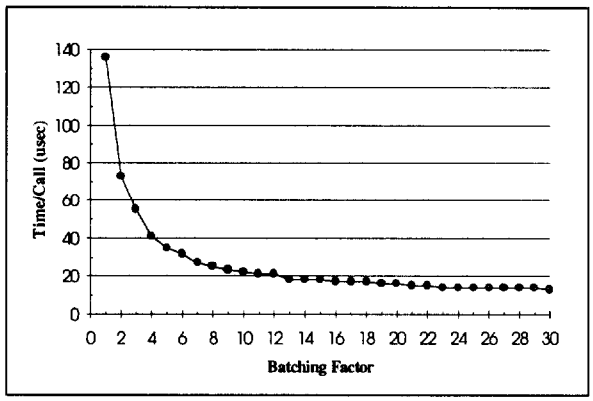

Figure 4: Local IPC.

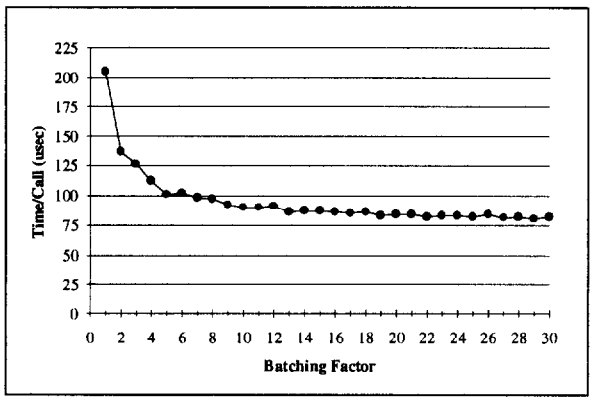

Figure 5: Local Thor

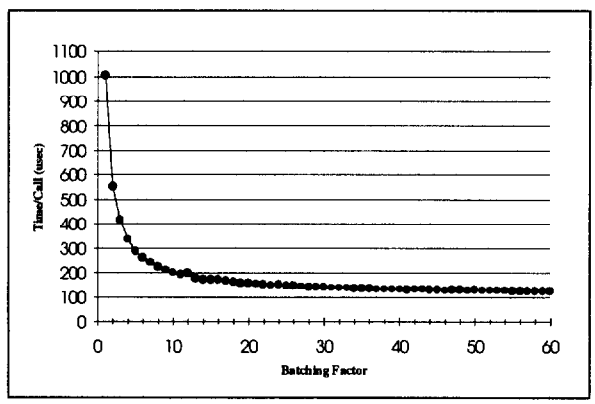

Figure 6: Remote Thor

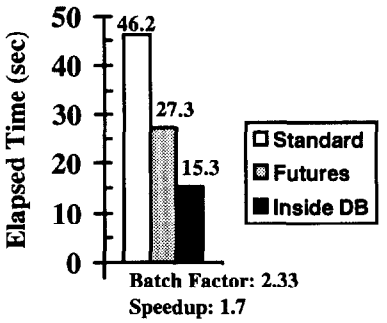

Figure 7: OO7 traversal performance.

an integer id or a character documentation string. We implemented the parts and their primitive operations as types in the Thor database, and the traversals in the Thor C++ veneer using the methods defined by the OO7 type interfaces.

In practice, one can safely increase performance by implementing parts of the traversal inside of the database using Theta. We wished, however, to assess the worst-case scenario (for our mechanism), in which the client performs a large number of very fine-grained interactions with the database, many of which need to be performed immediately.

We ran OO7 traversal 2b, the traversal that demands the greatest number of fine-grained database calls. The traversal visits every part in the database, swapping the x and y coordinates of each atomic part. It makes a total of 207,399 calls on the database and has an actual running time of 46.19 seconds, for an average of 222 microseconds per call.

The results of the experiment are shown in Figure 7. For this traversal the batching factor was 2.33. In other words, almost one out of every two operations was one that returned a basic value and forced the batch of calls to be flushed.

With this batching factor, we obtained a speedup of 1.7, giving us a running time of 1.8 times the optimal time for our current implementation. The optimal time was obtained by running the traversal when the client program was linked into the database, so that the domain crossing overhead was zero but all of the other costs were the same. (Note that the optimal time is about a third of the non-batched time, which matches our measurements in the list descent case that show t_c is about a third of $t_c + t_d$, the total overhead of running an operation without futures.)

However, getting so close to the optimal performance is largely an artifact of our unoptimized implementations of marshaling, type checking, and dispatching in the database. We estimate that the optimal time will improve by at least a factor of 3 when these implementations are optimized. In this case a batching factor of 2.33 only gets us a running time of 3.4 times the optimal performance.

Even with such low batching factors, batched futures yield useful performance improvements. The improvements are limited, however, by the batch size. One limiting factor in the current design is the need to stop batching as soon as a call returns a basic value. A simple way of addressing this limitation is discussed in the next section; a more sophisticated approach that can be used even when control structures depend on the results of Thor operations is discussed in Section 8.

7 Extensions

This section discusses two extensions to the future mechanism. First we discuss how to deal with exceptions; then we describe how to defer calls that return basic values.

7.1 Exceptions

In the normal case, Thor operations terminate by returning a value but they can also terminate by signaling an exception [12], indicating that a condition has occurred that prevents the normal return. An exception consist of a name and zero or more values that give additional information about the cause of the exception. For example, the *array* type might signal a *bounds* exception if the caller attempts to access a slot beyond the end of an array.

Exceptions represent a challenge for two reasons. First, many client languages fail to include any built-in mechanisms for handling them. Second, immediate detection of exceptions is incompatible with batching: if the veneer has to communicate with the database to

check for exceptions after every call, no batching is possible.

If the client language has an exception mechanism, it can be used to propagate information about Thor exceptions for non-deferred calls. Deferred calls, however, cannot signal exceptions, since they are not executed until later. Therefore, the stubs for deferred calls do not have exceptions listed in their signatures; instead such a stub always returns a stub object and we incorporate the information about the eventual outcome of the call (when it is known) in the stub object.

In addition, the database propagates exceptions. If a call that is supposed to produce a future signals an exception, the database stores information about the exception in the future's slot in F. If a later call uses that future as an argument, or uses an *error-handle* as an argument (see below), the database will not run the call but instead produces a *unhandled_exc* exception with a value that identifies the bad argument. (Thus *unhandled_exc* is a possible exception of any call.) When the database finishing processing a batch of calls, it returns all the information about exceptions associated with earlier calls in the batch in addition to the information about the outcome of the last call in the batch. The veneer then stores the exception information in the stub objects of the affected futures. If one of these stub objects is used in a subsequent call, the stub passes the error-handle in that argument position.

If the client language has no exception mechanism, we can use the above mechanism, except that something extra is needed for calls that return basic values (or return nothing). One possibility is to have such calls have an extra argument, an error value that the client code is supposed to check before using the call's result. This approach is error prone, however, since it is easy for programmers to forget to check for the error value. Therefore we use a different approach: we maintain an exception history, and store (effectively) information about the outcome of each call in it. This history is cleared automatically when a transaction starts; furthermore, it must have been cleared by the client code when a transaction commits, or the veneer will abort the transaction. Various operations are provided to give the client code convenient access to the history. More details about this mechanism can be found in [4].

The above approach allows the client to defer checking for exception values until convenient. In addition, it allows us to support exception handling while still achieving high batching factors.

7.2 Futures for Basic Values

An obvious extension to batched futures is the ability to use futures for operations that would ordinarily return basic values. Sometimes the client does not need to know basic value results immediately, if at all. For example, the client can swap the values of two slots in an array of integers without knowing what those values are, or sum a set of values without knowing the values of all of the intermediate results, as long as the database keeps track of the intermediate values and allows the client to refer to them using futures.

Futures in Multilisp [9] could be used transparently in place of basic values. However, this transparency came at a cost: the system had to add tags to every value to indicate whether it was a future or an actual value and every time an operation depended on the actual value of an object, it had to check the tag to see if the object was a future and if so block until the value was available. This approach is incompatible with our needs. Many client languages allow direct access to all of the bits of a value and will not tolerate tagging; furthermore inserting the necessary tag checks would require modifications to the client language compiler.

We therefore expose the distinction between futures for basic values and actual basic values to client programmers, allowing them to convert between the two forms on demand. Our scheme is like the promises mechanism used in Mercury [15] with the crucial difference that futures for values can be passed as arguments to Thor operations without blocking. Futures for values are distinguished from normal values by making them a distinct type. For each basic value type T, the veneer defines a corresponding stub type Thor-T; we shall refer to such types collectively as ThorValues. (For example, the stub version of a client integer is a *th_int*. Similarly, the veneer defines *th_char*, *th_float*, and so forth.) Conceptually, a ThorValue can be thought of as a pointer to a value that lives inside Thor. Each ThorValue type supports a "dereferencing" operation that

returns the corresponding client value and an operation that creates a ThorValue from a client value.

To allow futures for values to be passed as arguments to Thor method calls without dereferencing them, the veneer type interfaces include "futurized" versions of stub functions that take and return ThorValues rather than basic values. We also retain the standard versions of the stub functions. If the client wishes to batch a call containing a mix of client and Thor values, it must use a futurized stub, and must convert client values to obtain the ThorValues needed as arguments. As with handles, the veneer can batch a number of interrelated calls without communicating with the database. Unlike with handles, the client has a way to obtain the actual representation of the return result.

A ThorValue is represented in the veneer as a union containing either an actual basic value, or a future for a call that will return a basic value, with a tag to indicate which is the case. If the client attempts to dereference a ThorValue containing a future, the veneer sends the current batch of calls to the database.

To process a set of batched calls that use ThorValues, the database keeps a mapping between each future for a value and the corresponding basic value result, and sends back the actual results to the veneer in a batch. The veneer then overwrites the futures in the ThorValue stub objects with their actual values. Thereafter, the value for each of the ThorValues used in that batch of calls is available immediately without consulting the database.

A ThorValue needs to be *decoded* before it can be turned into a basic value [10] if the representation sent over by Thor differs from that used for the type in that language; also in this case a value will need to be *encoded* into a ThorValue. As an optimization, the veneer can store the value of the future in some 'raw' form when it first gets it and defer the decoding until the client actually asks for the value. This allows the veneer to save the expense of decoding and later encoding results that are only used as intermediate values by the client. For example, suppose the database and the client language use different formats for representing floating point numbers, and that it is expensive to convert between the two formats. If the client does not always need the values of the floating point numbers, the veneer could increase performance by performing the conversions lazily. (It might be advantageous to keep a copy of the original representation even after the number was converted, to avoid re-encoding the number if it is passed as an argument to another database operation.)

When futures are used for basic values, they are no longer completely transparent. The client must insert calls to obtain the client values corresponding to ThorValues when desired, and possibly also to wrap client values as database values. However, client programmers are always free to write their programs as usual with the normal version of calls, then convert them to use futures later as an optimization.

One concern is the proliferation of stub functions: adding futures for basic values has approximately doubled the number of stub functions. (The number is not exactly double because stub functions whose arguments and return values consist entirely of handles do not need a futurized version.) This can lead to a cluttered interface containing multiple versions of each stub function, which client programmers may find confusing. We can minimize this problem by choosing sensible conventions for naming and ordering stub functions so that the futurized versions are shown last. Figure 8 shows an interface using the conventions we have adopted: we capitalize the first letter of the method name to name futurized versions of stub functions. The figure also shows a client program that uses this interface to swap two elements of an array with no additional domain crossings.

We have not yet implemented futures for basic values. However, we calculated how they would increase the batch size for OO7 traversals and how that might affect performance. For example traversal 2b would benefit from the use of futures for basic values, because all calls to swap the x and y attributes can be deferred since the client has no need to know the actual x and y values. This increases the average batch size increase from 2.33 to 3.47 and the predicted performance using the mathematical model increases from 1.7 times the standard performance (for unoptimized Thor) to over 2 times faster.

```
class th_int_array {
  int      fetch(int slot) ;
  void     store(int slot, int val);
  ...
  th_int*  Fetch(th_int* slot);
  void     Store(th_int* slot, th_int* val);
};

void swap(th_int_array* a, th_int* i, th_int* j)
{
  /* Swap a[i] and a[j] using futures */
  th_int* a_i  = a->Fetch(i);
  a->Store(i, a->Fetch(j));
  a->Store(j, a_i);
}
```

Figure 8: Using futurized stubs.

8 Conclusion and Future Work

This paper has described batched futures, a mechanism that reduces the delay to clients making calls to servers that run in a separate protection domain. Futures allow calls to be deferred until a client really needs a result. Then all calls can be made at once, in a batch; later calls in the batch can refer to the results of earlier calls. The paper analyzed the performance gains that can be expected from the mechanism. Our results show that significant benefit can be obtained by using futures even when batches are small.

The work was done in the context of Thor, but can be used in other systems. Our implementation depends on knowing whether a call returns an opaque pointer or a value and can be used in any system where this information is available, for example, an operating system. Even without this information, an approach in which the client chooses whether or not to defer a call is always possible. The mechanism can be used when the client and server run in different processes on the same machine, and also when the client runs at a different machine than the client.

Speedups are limited by the number of operations that can be deferred. For example, the OO7 benchmarks do not allow deferring very many calls, even with futures for values, because after a few calls the application needs to do something itself, e.g., look at a value to determine what to do next. One possibility we are investigating is to enrich the "batching language" used to communicate with the database. Currently, the batched program can include interrelated calls, but nothing else. Much larger batching factors could be achieved if the client were allowed to combine calls with simple batched control structures; for example, the entire traversal 2b could be done in a single batch. To the client, batched control structures appear much like normal control structures. Instead of affecting the control flow of the client program, however, the batched control structures are interpreted in the database, which carries out calls only on the indicated paths. Because a batched loop needs to be analyzed and type checked only once by the database, batched control structures amortize not only the expense of domain crossing but also that of processing each call. The effect is somewhat like what is achieved with query languages. However, in this case the user does not need to learn any special query language – the "query" is constructed on the fly based on the batched calls and control structures used by the client.

Another possibility is suggested by Stamos' work on *remote evaluation* [16]. The idea is to define a "safe" subset of the client language such that procedures written in that subset are guaranteed not to violate the security of the database. The compiled code for such a procedure can then be installed in the database, and calls to it can run inside the database. Probably not all languages have useful safe subsets, but looking for them, and defining preprocessors that recognize them, is an interesting research direction. If an entire application could be written in the safe language subset, it could run inside the database without any domain crossings, thereby achieving the best possible performance.

Acknowledgements. The authors gratefully acknowledge the helpful comments of Atul Adya, Mark Day, Umesh Maheshwari, Andrew Myers, James O'Toole, Quinton Zondervan, and the referees.

References

[1] Barrera, J., Invocation Chaining: Manipulating Lightweight Objects across Heavyweight Boundaries. In *Fourth IEEE Workshop on Workstation Operating Systems*, Oct. 1993, 191-193.

[2] Bershad, B., Anderson, T., Lazowska, E., and H. Levy, Lightweight Remote Procedure Call. In *ACM Transactions on Computer Systems*, Vol. 8, No. 1, Feb., 1990, 175-198.

[3] Birrell, A.D. and B. J. Nelson, Implementing Remote Procedure Calls. In *ACM Transactions on Computer Systems*, Vol. 2, No. 1, Feb. 1984, 39-59.

[4] Bogle, P., *An Efficient Object-Database Interface using Batched Futures*. Lab. for Computer Science Tech. Report TR-624, MIT LCS, Cambridge Ma., July 1994.

[5] Carey, M. J., DeWitt, D. J., and J. F. Naughton, The OO7 Benchmark, In *Proc. of the 1993 ACM Sigmod International Conference on Management of Data*, ACM Sigmod Record, Vol. 22, No. 2, June 1993, 12-21.

[6] Day, M., *Client Cache Management in a Distributed Object Database*. PhD. Thesis, MIT, Cambridge, Ma., forthcoming.

[7] Day, M., Gruber, R., Liskov, B., and A. Myers, *Abstraction Mechanisms in Theta*. Prog. Method. Group Memo 81, Lab. for Computer Science, Cambridge Ma. forthcoming.

[8] Gifford, D.K. and N. Glasser, Remote Pipes and procedures for efficient distributed communication. In *ACM Transactions on Computer Systems*, Vol. 6, No. 3, Aug. 1988, 258-283.

[9] Halstead, R., Multilisp: A Language for Concurrent Symbolic Computation. In *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, Oct. 1985, 501-538.

[10] Herlihy, M.P. and B.L. Liskov, A value transmission method for abstract data types. In *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, Oct. 1982, 527-551.

[11] Liedtke, J., Improving IPC By Kernel Design. In *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles*, ACM Sigops Review, Vol. 27, No. 5, Dec. 1993, 175-187.

[12] Liskov, B.L. and A. Snyder, Exception Handling in CLU. In *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 6, Nov. 1979, 546-558.

[13] B. Liskov, Day, M., and L. Shrira, Distributed Object Management in Thor. In *Distributed Object Management*, M. T. Ozsu, U. Dayal, and P. Valduriez, Eds., Morgan Kaufmann, 1992, 79-91.

[14] Liskov, B.L., Communication in the Mercury System. In *Proc. of the 21st Annual Hawaii Conference on System Sciences*, IEEE, Jan. 1988, 178-187.

[15] Liskov, B. L. and L. Shrira, Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proc. ACM Sigplan '88 Conference on Programming Languages Design and Implementation*, ACM Sigplan Notices, Vol. 23, No. 7, June 1988, 260-267.

[16] Stamos, J., *Remote Evaluation*, PhD. Thesis, Lab. for Computer Science Tech. Report TR-354, MIT LCS, Cambridge, Ma., Jan. 1986.

[17] Wahbe, R., Lucco, S., Anderson, T., Graham, S., Efficient Software-Based Fault Isolation. In *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles*, ACM Sigops Review, Vol. 27, No. 5, Dec. 1993, 203-216.