Integrating support for undo with exception handling

Avraham Shinnar[1], David Tarditi, Mark Plesko, and Bjarne Steensgaard

December 2004

Technical Report
MSR-TR-2004-140

One of the important tasks of exception handling is to restore program state and invariants. Studies suggest that this is often done incorrectly. We introduce a new language construct that integrates automated memory recovery with exception handling. When an exception occurs, memory can be automatically restored to its previous state. We also provide a mechanism for applications to extend the automatic recovery mechanism with callbacks for restoring the state of external resources. We describe a logging-based implementation and evaluate its effect on performance. The implementation imposes no overhead on parts of the code that do not make use of this feature.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

[1]The work by Avraham Shinnar was done during an internship at Microsoft Research.

1 Introduction

Exception handling is used for error handling in many programming languages. A problem with exception handling is that programmers are responsible for restoring program state and invariants and freeing resources when an exception occurs. This can be hard to implement correctly and be difficult to test since programs may fail at unexpected points or in unexpected ways. This results in programs that have error handling code, but when errors occur either fail immediately or eventually.

To help improve program reliability, we propose adding to imperative programming languages an undo feature that is integrated with exception handling. The undo feature automates the restoration of program state after an exception occurs. In this paper, we describe the design of this feature and discuss initial implementation and optimization results. We have implemented this feature in Bartok, an optimizing ahead-of-time research compiler and runtime system for Common Intermediate Language (CIL) programs, and have extended the C# programming language with this feature. We evaluate the implementation using programs that include microbenchmarks, C# versions of SPEC95 benchmarks, and optimization phases from the Bartok compiler. We demonstrate that this feature can be used in imperative code. Code speed ranges from 28% to 100% of the speed of the original code. There is no change in speed for code that does not use this feature.

A code block can be declared to be a try_all block. If an exception occurs in the code that is not caught before reaching the scope of the try_all, the program state is restored to the value that it had when execution of the block began. Try_all blocks can contain arbitrary code and can be nested. To support automatically freeing resources and restoring state associated with the external world, i.e. libraries and the operating system, programmers can register "undo" functions during program execution. If state rollback occurs, these functions are invoked in the reverse order that they were registered. The design allows external libraries and calls to be wrapped, so that the undo code needs to be written only once.

We focus on the design of undo with exception handling for single-threaded programs. We assume that for multi-threaded programs, some other mechanism is used to prevent data races. A concurrency control mechanism for preventing data races is beyond the scope of this paper.

Our implementation supports all the features of CIL, including virtual calls, garbage collection, stack-allocated structs, pointers, and unsafe code. The implementation keeps a stack of lexically active try_all blocks. Within each try_all block, the old values of memory locations are logged before the memory locations are modified. This is amenable to optimization because the original value of a memory location needs to be recorded only once within an undo block. To avoid imposing a logging cost on code that does not use undo, the compiler clones and instruments code reachable from try_all blocks. We assume that we have the whole program.

To achieve adequate performance, the implementation incorporates a number of optimizations. First, we organized the logs as linked lists of arrays, so that logs could grow without incurring a copying cost. Second, we modified the generational copying collector to treat logs as root sets for garbage collection to eliminate write barrier checks. Third, the compiler uses a static analysis to avoid logging stores to objects that are newly allocated in an undo block. Fourth, the runtime system detects writes to memory locations that have been logged previously in the current undo block.

The rest of the paper is organized as follows. Section 2 describes the semantics of undo and provides some examples. Section 3 describes the basic implementation of undo and Section 4 describes further optimizations. Section 5 evaluates the implementation. Section 6 discusses related work.

2 Semantics

To support automatic error recovery, we propose adding a new keyword, try_all, to the C# programming language. Syntactically, try_all has identical usage as the existing try keyword. Try_all blocks behave similarly to try blocks, with the added feature that they restore memory back to its initial state if an exception reaches the scope of a try_all block. We first present the basic semantics of this mechanism and then discuss extensions to support more advanced functionality.

2.1 Basic Semantics

The semantics of try_all can be thought of as a memory checkpoint followed by a try block, with the added property that if an exception is propagated up to the scope of the try block, memory is reverted back to the checkpointed state.

2.1.1 Motivating Example

In the example shown in Figure 1, an object is moved from one queue to another. If an error happens while

```
void move(Queue to, Queue from) {
  try_all {
     to.Enqueue(from.Dequeue);
  }
}
```

Figure 1: Moving an item from one queue to another with transparent recovery

```
void move(Queue to, Queue from) {
  object obj = from.Dequeue();
  try {
    to.Enqueue(obj);
  } catch(Exception ex) {
    // try to add the object back to the old queue
    try {
      // most queues don't even support this
      // operation as it breaks queue semantics.
      from.AddToBack(obj);
    } catch(Exception ex) {
      // what do we do now? obj is lost
      System.Exit(-1);
    }
  }
}
```

Figure 2: Moving an item from one queue to another without transparent recovery

```
try_all {
  // code that may throw an exception
} catch(ExceptionSubClass ex) {
  // this code runs before memory is reverted
  // and is itself reverted
}
// ... other catch handlers
catch {
  // this code runs before memory is reverted
  // and is itself reverted
} finally {
  // this code runs after memory is reverted
  // and effects of the code are not reverted
}
```

Figure 3: Try_all Syntax

adding the item to the "to" queue, memory is automatically reverted by the try_all mechanism, and obj is restored to its original place in the "from" queue. The caller of the move function can then try to recover, knowing that at least the data is consistent (and has not been lost).

Compare this with the example shown in Figure 2 of the same function coded without the use of the try_all keyword. Attempting manual error recovery is considerably more complicated. If something goes wrong after removing obj from the "from" queue, it may not be possible to return obj to its original location.

The distinguishing feature of this example is the interdependence of the two queues. More generally, a key motivation for designing this language extension was code that modifies interdependent data structures and needs to keep them consistent in the face of errors.

2.2 Integration with exception handling

Try_all blocks are syntactically identical to try blocks, except for the leading keyword. They can be fully nested within each other and within try blocks. Like try blocks, try_all blocks may have associated catch handlers. Catch handlers associated with a try_all block are run in the context of the try_all block. In the event of an exception, the corresponding (as per the usual C# definitions) catch handler is run, after which the try_all block (including the catch handler) is aborted. If the catch handler returns normally, control resumes, after the try_all block is aborted, after the try_all block. If the catch handler throws a new exception or rethrows the original exception, the

exception continues propagating up, after the try_all block is aborted, looking for catch handlers associated with higher-level try or try_all blocks.

Try_all blocks may also have associated finally handlers. These are executed after the try_all block and outside of its context. So, in the case of an exception, the try_all block is aborted first before the finally handler is run as usual. The general syntax is shown in Figure 3. Note again the similarity to the normal syntax used by try blocks.

Exception objects that are created inside of a try_all block must be allowed to propagate outside of the try_all block without being reverted. To do this, exceptions leaving the scope of an aborted try_all block are cloned before memory is reverted. Because it difficult for the system to know how deeply to clone an exception object, we add a protected cloneForUndo method to the Exception base class. This allows Exception classes to specify how they should be cloned. This method is called in a context where logging is disabled by the NoLoggingForUndo attribute (see Section 2.3), ensuring that the new Exception object it creates and returns is not reverted by the try_all mechanism. As with all NoLoggingForUndo methods, this method must be coded carefully.

2.3 Tracing and Profiling: NoLoggingForUndo

In many applications there are operations that the programmer does not wish undone when an error happens. Two common examples are tracing and profiling. When an error occurs, the programmer would like to preserve the information already gathered, especially if the programmer is gathering information about what happens in exceptional circumstances. To support this, we allow programmers to mark methods as being unaffected by any enclosing try_all block. We use the C# attribute mechanism to do this. We define a new attribute NoLoggingForUndo that can be used to mark methods and classes. Modifications to memory in these methods and classes are not undone.

When a method marked as NoLoggingForUndo calls another method, that method is executed in a NoLoggingForUndo context, no matter how it was marked. Because of this, programmers need to be careful about calling methods that may modify nonlocal memory from NoLoggingForUndo methods.

2.3.1 Example

Figure 4 shows an example use of NoLoggingForUndo. The figure contains code from a class that

```
class ExceptionTracker {
  [NoLoggingForUndo]
  public void Add(Exception ex) {
    exList.Add(clone(ex));
  }
  private Exception clone(Exception ex) {
    // clone the exception
    ...
  }
  ...
  private ArrayList exList = new ArrayList();
}
```

Figure 4: Keeping a list of exceptions

keeps a list of exceptions that have occurred. The intended usage is that there will be a globally accessible object of type ExceptionTracker and that the Add method of this object will be called by all exception handlers.

In this example the Add function stores a clone of the exception, rather than the passed in reference. Because the clone is created within the context of Add, a NoLoggingForUndo method, the clone is protected against having its contents being reverted by a try_all block.event of an exception.

2.3.2 Interaction of NoLoggingForUndo with try_all

Consider what happens if NoLoggingForUndo code and normal code in a try_all block write to the same memory location and the try_all block aborts. It is unclear what state memory should be put into. The obvious choices are: the state before the try_all, the state after the NoLoggingForUndo method completes, and its current state (don't change it). Three possible ways to handle this are:

1. Declare that it leads to undefined behavior. Type safety is maintained, but the state of memory after a try_all block aborts may be any of the values that it has assumed since the start of the try_all block (including its original state).

2. Dynamically check to make sure that this situation never occurs. If it does, signal an error. The difficulties with this are that it would be expensive to check all writes to memory and that it could cause an unexpected failure in a program.

3. Statically ensure that this situation can't arise. The simplest method would be to use separate memory pools.

```
public interface IUndo {
  void Undo();
}
```

Figure 5: IUndo interface

We chose the first option in our implementation: overlapping writes lead to undefined behavior. This avoids the runtime overhead of dynamic checking. Static checking via separate memory regions would probably be a better choice if it were supported by CIL, but it is not.

2.4 Dealing with the external world

Applications are rarely self-contained: they generally need to obtain resources from the operating system and request that the operating system change the state of these resources. A common example is files. Many programs obtain a file handle from the operating system and then invoke system calls to read, write, and seek around in that file. Although some file systems are transactional, and thus support undo semantics, most are not. As a result, it is necessary to provide a mechanism for allowing the user to specify how to undo the effects of system calls. In theory, the catch handlers associated with try_all blocks can be used for this, however this requires that users of libraries that make system calls know how to undo their effects. A more modular solution is to allow a library itself to specify how those system calls are undone.

2.4.1 Recovery Call-backs: RegisterForUndo

To deal with this problem, we allow the programmer to register call-back methods with the undo system. If a try_all context containing the point of registration is reverted, the registered call-back will be invoked to allow reversal of external state. The RegisterForUndo method may be called at any point with an argument object implementing the IUndo interface shown in Figure 5. In the event of a try_all block aborting, the IUndo derived objects registered will be called in the reverse order of their registration. When they are called, memory will have been reverted to the state it was in at the time the object was registered. This allows the writer of IUndo derived classes to easily store references to needed information with the assurance that the memory pointed to by those references will be in the same state in case of an abort, regardless of later writes to memory.

```
class SeekUndo : IUndo {
  public SeekUndo(File f, int bytes) {
    this.f = f;
    this.bytes = bytes;
  }

  public void Undo() {
   f.Seek(-bytes);
  }

  private File f;
  private int bytes;
}
```

Figure 6: SeekUndo class

```
class SmartFile {
 ...
 int ReadInt() {
   int ret = f.ReadInt();
   TryAllManager.RegisterForUndo(new SeekUndo(f, 4));
   return ret;
 }

 ...
 private File f;
}
```

Figure 7: Smart File Wrapper.

2.4.2 Example

In Figure 7 a fragment of a SmartFile class is presented. This class is intended as a wrapper for the File class. After invoking any system calls, such as ReadInt, it registers an Undo object that can undo the effects of the system call. In this case, the undo object is an instance of the SeekUndo class, given in Figure 6. This object is capable of seeking backwards in the file to restore the file position to the state it was in before the read.

```
try_all {
   undoObject.Undo();
   throw new FakeException();
}  catch(FakeException) {
}
```

Figure 8: Undoing the Undo

2.4.3 Undoing an undo

Strange as this may seem, the memory changes of the Undo method are themselves undone. When a try_all block detects an error and starts reverting memory, the registered objects' Undo methods are invoked. In our implementation the methods are invoked within the context of a new try_all block, which is guaranteed to terminate with an exception, as is illustrated in Figure 8.

Undo methods are themselves undone so they do not break the semantics of nested try_all blocks by modifying memory. To resynchronize the state of external resources with (reverted) memory, Undo methods may need to call methods that modify the state of memory while setting up various system calls. By invoking the Undo methods inside of try_all blocks that always abort, we can allow programmers to use arbitrary methods from within an Undo method without having to worry about the effect of those methods on memory.

2.4.4 When an undo method throws an exception

Because Undo methods can run arbitrary code, they may throw exceptions. Defining a reasonable semantics for this case is difficult because this is a case of an error handler failing. Three choices are:

1. Ignore the exception.
2. Terminate the Undo in progress. This would leave the program in an undefined state. It would, in general, be extremely difficult for the application to recover at this point.
3. Keep a list of failures and propagate this information to other Undo handlers. When the try_all block finishes aborting, this list could be made available to the application via a special handler attached to the try_all block.

Our implementation ignores exceptions thrown from Undo handlers. It is expected that Undo handlers are to be coded defensively and should not throw exceptions. This keeps the design simple and understandable without imposing an unreasonable burden on the programmer.

2.5 Concurrency

Concurrency control mechanisms and try_all blocks should be able to coexist without a problem. The main effect of the try_all blocks from a concurrency standpoint is that if an error occurs, the try_all mechanism may write to any memory written to in the context of the try_all block in order to return memory to its prior state. So a locking scheme, for example, would need to ensure exclusive access to memory accessed in try_all contexts.

3 Basic Implementation

We implemented try_all support in Bartok, an optimizing ahead-of-time research compiler and runtime system for CIL. Bartok takes one or more CIL files and generates stand-alone native x86 executable programs. Bartok is the successor compiler to Marmot [4].

We extended the runtime system to store and manipulate state needed by try_all. The data is stored in normal, garbage collected memory, simplifying the implementation. We also extended the compiler to understand try_all blocks and to instrument the program to support them. Like most of Bartok, try_all support is implemented entirely in (mostly safe) C# code.

3.1 Goals

We had the following implementation goals:

- Code that does not use try_all blocks should not be slowed down by try_all blocks.
- Try_all blocks should be cheap to enter and exit normally. We want programmers to be able to use them for small blocks of code without worrying about large overheads.
- Try_all blocks should support all the language features available in C#, including pointers, unsafe code, stack-allocated structures, and virtual calls.
- We should allow the use of common garbage collection techniques such as copying collection, with minimal changes for the sake of efficiency.
- Code should be executed efficiently within a try_all block.

3.2 Overview

Our initial implementation uses logging. Inside a try_all block, changes to memory are logged. If a try_all block needs to revert memory, it uses these logs to restore memory to its original state. The implementation has three high-level parts:

- The compiler inserts code to log the original value of a memory location before it is changed within a try_all block. If a try_all block fails, the runtime system uses the log to set memory back to its original state.
- The runtime system keeps a stack of active try_all blocks. When a try_all block begins, the runtime system records the point in the log where the block began.
- If a try_all block ends normally, it is popped from the stack of try_all blocks. The log keeps growing until the outer-most try_all block is popped, at which point the log is cleared.

3.3 Rejected Alternatives to Logging

The logging approach allows for a simple implementation of nested try_all blocks. Similarly, it also simplifies ensuring that Undo functions are invoked with memory in the same state as when they were registered. Because logging is by nature incremental, it is relatively simple to revert to an intermediate memory state.

We considered two alternative approaches for allowing memory to be reverted:

1. Checkpoint the state of memory when entering a try_all block. When a try_all block exits normally, release the checkpointed memory. When an exception occurs, revert memory to its checkpointed state.
2. Inside of a try_all block, write memory modifications to a different address space. Attempts to access data will read from the new space if the data is there, otherwise they will read from the old space. When a try_all block exits normally, the two address spaces are merged together. When an exception occurs, the new memory space is discarded.

Because one of our goals is to allow try_all blocks to be cheap to enter and exit normally, the first choice, checkpointing all of memory, is not practical. The second choice imposes an overhead on every memory read. The overhead will be proportionally greater when only a small fraction of each page is modified. Because of these problems, we chose the logging based implementation.

3.4 Logging: Runtime Support

This section discusses the runtime support for logging. The compiler modifies code that can (directly or indirectly) run inside of a try_all block by injecting logging instructions before operations that modify memory.

```
void mutator(ref int x) { // call with a reference
   x = 3;
}
class IntWrapper {
  int i = 5;
}
void call_mutator() {
  IntWrapper heap_int_wrapper = new IntWrapper();
  int stack_int1 = 6;
  try_all {
    int stack_int2 = 6;
    mutator(ref heap_int_wrapper.i); //first call
    mutator(ref stack_int2);  // second call
    mutator(ref stack_int1);  // third call
    throw new Exception(); // force an error
  }
  ...
}
```

Figure 9: Stack v. Heap Memory

3.4.1 Log Structures

There are actually four different arrays that store logging information. Where information goes depends on the type of the memory. Memory is split up into the following categories:

1. Memory has value type and is on the heap.
2. Memory has reference type and is on the heap.
3. Memory has value type and is not on the heap.
4. Memory has reference type and is not on the heap.

It is important to differentiate between memory that is on the heap and memory that is not on the heap. In particular, memory that is on the stack must be differentiated from memory that is on the heap. Figure 9 shows an example illustrating the need for this distinction. The mutator method must log its modifications as it is called in a try_all context. But, if we always restore the old value of memory when the try_all block aborts, the second invocation of the mutator method may lead to a problem. The memory that was changed may no longer be on the stack frame and writing to it may be illegal. We thus have to check to see if stack based memory is still valid. As we see from the third invocation, we still have to log

(and sometimes restore) stack based memory. As a result of these considerations, we maintain separate arrays for heap and non-heap memory.

Because C# is a garbage collected language, it is important that the try_all runtime system interact well with the garbage collector. Many garbage collectors need to differentiate between GC managed pointers, which they trace, and unmanaged data, which they do not trace. For that reason, we differentiate between memory representing a value type from memory representing a (traced) reference type.

log structures used to store this data.

Interior Pointers Many garbage collectors either do not support storing interior pointers in heap data structures or impose an additional cost associated with their use. For these reasons, the logging structures store references to the base object and offsets into it, rather the interior pointer to the modified memory. These values are easily computable at the point where a store to memory is being done.

Independence of the Arrays Although there are four different logging arrays, it is important to realize that a given memory location will always be mapped to the same array. This frees the runtime system from having to timestamp entries. A memory location is mapped consistently because the properties that determine which array it belongs in do not change over time. Clearly, whether or not a memory location is on the heap will not change. Additionally, the same memory will not be accessed both as a value type and as a reference type. Doing so will cause problems with the garbage collector.

3.5 Logging: Compiler Transformations

All operations that can modify memory may potentially be logged, if they are executed within the context of a try_all block. It is easy to determine if an operation is lexically enclosed within a try_all block. If it is, then an appropriate log operation is injected into the instruction stream before the operators.

These simple changes suffice for code that is lexically contained within a try_all block. However, try_all blocks may invoke arbitrary methods. Invoked methods must also be modified appropriately to log any modified data. However, one of the goals of this implementation is that code not running within a try_all block should not incur any overhead. Modifying all methods to support logging would add the overhead of logging (or at least a test) to that code.

3.5.1 Method Cloning

To avoid this problem, methods transitively called from within a try_all context are cloned. The cloned methods are modified to support logging. All calls to the methods made from within a try_all context are changed to instead call the instrumented cloned methods. Method invocations within any of these clones are dealt with similarly. Note that only one clone of a method is ever created; all invocations from a try_all context call the same modified clone. If a method is marked as NoLoggingForUndo (see Section 2.3), the method is not cloned and the call site is left unchanged.

Virtual Methods Virtual methods complicate the cloning of methods. A single method cannot just be cloned, because, at runtime, an override of the method may be called instead. We instead perform an analysis to determine all overrides of a given method. We then clone all of the overrides as well as the method that is explicitly being invoked.

3.5.2 Local Variables

Changes to local variables, for the most part, do not have to be logged. The compiler inserts code that saves the values of all local variables live upon abnormal exit from a try-all block. The code for aborting a try-all block restores the values of those local variables from the saved values.

3.6 Try_all Blocks: Runtime Support

The runtime system stores a stack of try_all block records. Each one keeps markers into the logging arrays that indicate when they started. This allows for easy support of nested try_all blocks. The runtime support has methods to begin and terminate try_all blocks.

3.6.1 Start

The Start method starts a new, possibly nested, try_all block. It creates a new block record, sets the log array markers to point to the current end of each of the log arrays. It then pushes the new record onto the try_all blocks stack.

3.6.2 Commit

The Commit method pops the top record off of the try_all block stack. If it was a top-level try_all block (the stack is empty; note that in our implementation

this is a dynamic runtime concept, not a static lexical one), it also frees the memory associated with the log arrays and the IUndo registry. Note that if it is not a top-level block, the parent try_all block inherits the log data by popping the record off. If the parent try_all block later aborts, it will need to revert memory that was modified during this block, even though this block committed successfully.

3.6.3 Abort

The Abort method is the most complicated of these methods. It is called when an exception reaches a try_all block, and is responsible for reverting memory back to its original state. It then pops the top record from the try_all block stack, just like commit.

Abort goes backwards through the four log arrays, restoring the original values of modified memory locations using the values preserved in the logs. It does this in each array until it reaches the corresponding marker stored in the block record.

This basic algorithm is complicated by the possible presence of registered IUndo derived objects. See Section 3.8.1 for details.

3.7 Try_all Blocks: Compiler Transformations

The compiler injects a call to Start at the beginning of the try_all block in order to create a new block. It then inserts a Commit call right before each normal exit from the try_all block. It modifies the catch handlers associated with the block to call Abort right before any normal exits. It also wraps the whole try_all block and its associated catch handler in the equivalent of a new try block, which has a provided catch handler. This handler catches any exceptions, clones the exception, calls Abort, and, finally, throws the exception clone.

3.7.1 Cloning the Exception

If a try_all block is being exited via an exception, the runtime system clones the exception as described in Section 2.2. Specifically, the runtime system calls the cloneForUndoMethod for the exception class, which by default does a shallow copy.

3.8 IUndo

To support the registration and invocation of IUndo derived objects, the runtime system stores an array of undo records. Each record stores a registered object, as well as markers into the logging arrays. This allows the system to track when the object was registered.

3.8.1 Modifying Abort

The abort algorithm of Section 3.6.3 must be modified to support registered objects during a try_all abort, Before the IUndo.Undo method of a registered object is invoked, memory is reverted back to the state it was in when the object was registered. This is done by going backwards through all four logging arrays until the markers stored in the undo record are encountered. The IUndo.Undo method is then invoked. If there are more registered IUndo objects for the try_all block, Abort continues going backwards through the arrays until it encounters either the place for the next IUndo object was registered or the beginning of the block (as determined from the markers).

3.8.2 Calling IUndo.Undo

The IUndo.Undo method is not invoked directly, but rather via a helper function that wraps the invocation in a new try_all block that always aborts, as required by the design (described in Section 2.4.3).

Note that Abort will be called recursively, since if Abort calls the Undo method on a registered IUndo object, it will in turn abort the try_all section that is created, as was just described. This in turn calls the Abort method.

3.9 Concurrency

All of the runtime data structures described above are stored on a per-thread basis. Operations on these data structures reference the appropriate thread-local versions. Thus, the notion of a point in time in the log structures, as described above, makes sense, since logs are relative to a single thread, which has a single stream of execution.

With our current implementation, programs must implement a higher-level policy for concurrency control. Specifically, for locking, any locks needed by a given thread should be acquired before entering a try_all block and should be released afterward.

4 Further optimizations

Further improvements to the implementation are divided into three areas: improving the logging implementation, eliminating unnecessary logging operations at compile time, and eliminating unnecessary logging data at run time.

4.1 Improving the logging implementation

Instead of implementing each log as an array, as described in the previous section, we implement each log as a linked list of array buckets. When a log fills, a new array is allocated and the previous array is added to the linked list. The cost of this is amortized by choosing a suitable array size (currently 128 elements, although we could use an adaptive strategy). We modified the runtime operations that used an index in the log to instead use a pair consisting of the array and an offset into the array. Using a linked list of arrays is more efficient than using a single array to represent the log. When the single array fills, a new array has to be allocated and the contents of the original copied to that array.

We also modified the garbage collector to treat the log data specially. In this paper, we use a generational copying collector in the Bartok runtime system.[1] We treat the log data specially because each log element contains a pointer to the object being modified. In addition, log elements for memory locations containing reference types contain pointers to the original reference (object) type. Without special treatment, each logging operation needs one or two write barrier operations in a generational collector.

The generational collector treats the log data for each thread as part of the root set when a collection of the nursery is done. To scan only the new log data added since the previous garbage collection, the system maintains a low water mark for each log. The low water mark is the earliest point in a log that has been modified since the previous collection. It is easy to incorporate this tracking into the try_all block operations described in Section 3.6.

4.2 Eliminating logging operations at compile time

Logging has three interesting properties from a compile-time optimization perspective. First, modifications to data that are allocated since a try_all block began do not have to be logged. Note that if the data are modified within a nested try_all block, the data does not count as newly allocated since that try_all block began. Second, only the first modification to a memory location in a try_all block has to be logged. Third, it is correct to log a memory location earlier than necessary, as long as the logging is done after the enclosing try_all begins.

[1]The Bartok runtime system has several garbage collector implementations

To take advantage of the first property, we have an interprocedural analysis that tracks newly-allocated data and eliminates logging operations on stores to that data. The analysis tracks when a variable is bound to newly allocated data. The analysis is flow-sensitive within a function and flow-and-context insensitive interprocedurally. The information about data being newly allocated is killed at the beginning of a try_all block.

It is possible to take advantage of the second and third properties by modifying code motion and redundancy elimination optimizations. The rules for code motion are simple: a logging operation cannot be moved before a try_all block begins or after a try_all block ends. In addition, a logging operation cannot be moved after any modifications to the memory location being logged. The modifications to redundancy elimination are similarly straightforward: logging operations in a try_all block that are dominated by other logging operations that must log that memory location can be eliminated. Even further optimizations are possible by combining these optimizations with object-level logging, where an entire object is logged instead of just a single location within the object.

4.3 Eliminating unnecessary logging data at run time

A simple implementation of logging will use space proportional to the number of times heap data is modified in a try_all block (i.e. proportional to running time), as opposed to space proportional to the amount of data modified. For long-running try_all blocks, this can be problematic. Thus, it is important to have strategies for eliminating redundant log entries or compressing log data.

The problem of detecting redundant log entries is similar to the problem detecting redundancy in remembered sets for generational garbage collectors, which has been well studied. A remembered set tracks all pointers from older generations to a younger generation.

We chose to use a hash-table based filtering mechanism that tracks the memory locations that have been logged recently. The filtering is done as part of the logging process. The hash function for memory locations is cheap to compute. When collisions occur, we overwrite the hash table entry with the most recent memory location. One issue is that garbage collection, which may move data, invalidates the hash table. We currently allow the hash table to be repopulated and do not rehash existing log entries when this happens. The default size of the table is 2,048 entries. It can be overriden manually on a per-application

basis. Obviously an adaptive scheme that monitors misses in the cache and the benefit from increasing the cache size could be used to determine the appropriate hash table size.

5 Evaluation

In this section, we first evaluate our implementation qualitatively and then evaluate it quantitatively.

The performance of a try_all implementation depends on several factors: the amount of data live on entry at the beginning of a try_all, the number of heap writes as a proportion of instructions executed during a try_all, the number of distinct heap locations written during the try_all, and the amount of data allocated during the try_all. A try_all implementation must retain all data live on entry at the beginning of a try_all block, unless it uses an escape analysis to determine data that is dead after the try_all.

Our implementation will do well for short-running try_all blocks, try_all blocks not containing too many heap writes, or try_all blocks that write to a small number of heap locations.

Our implementation may not do well for long-running try_all blocks or try_all blocks that modify many heap locations. It will retain most data allocated in a try_all, unless the data is used only in initializing writes (that is, in a functional style). It will also need space for log entries for each distinct heap location that is written.

5.1 Benchmarks and experimental setup

Table 1 shows the benchmarks that we used to evaluate the logging implementation. The programs in the first section (compress through wc) are originally Java programs from the IMPACT/NET project [12, 11]. They were hand-translated to C#. The second section includes the Bartok inlining phase and an implementation of quicksort. As described in Table 1, we modified each benchmark to use try_all. This is crucial choice in evaluating try_all. We purposefully chose to place try_all blocks to enclose large parts of the benchmarks to stress our implementation.

Our most realistic benchmark is the Bartok inliner. The Bartok inline phase does a bottom-up traversal of the call graph, inlining into each method. If it does inlining in a method, it calls various per-function clean up optimizations. We modified the inline phase to wrap the processing of each method in a try_all block. One might do this because of concern about optimizations or the inliner itself failing.

Because the inlining phase makes extensive use of other compiler components such as the IR and clean up opimizations, we only present a conservative estimate of the line count. We are able to self-compile the Bartok compiler using a pass of inlining where each per-method inline is undone using try_all.

We measure performance on a Dell Precision WorkStation 450. The processor is a 3.06 Ghz Xeon (tm) with an 8Kbyte first-level cache and a 512 KByte second-level cache. It has 2 Gbytes of RAM The operating system is Windows XP Professional (tm) with Service Pack 2. The version of the CLR is 1.1.4322.

5.2 Effect of logging on program speed

Table 2 shows the effect of logging on program speed. The first column is the speed of the original programs when run by the CLR. The second column is the speed of the original programs when compiled by Bartok and run as stand-alone executable. The third column shows the speed of the programs modified to use try_all. All columns are normalized to the speed of the original programs when compiled by Bartok (the second column). The speed of the CLR versions of these programs demonstrates that Bartok produces code competitive with a production system.

We see that our initial logging implementation has a wide range of effect on program speed, ranging from no effect (for othello) to reducing speed to 28% of the original speed (for compress). Othello is an example where filtering works very well: the program's state consists of small game board. There is no effect of logging on performance.

Compress and linpack are benchmarks where logging individual memory writes works poorly. Each benchmark is modifying a large in-memory array. The entire array or object could be logged before executing the core loops of these programs. This suggests pursuing a strategy that uses programmer hints or profiling to decide what kinds of log operations to place and where to place them.

For our most realistic benchmark, the Bartok inliner, we found that its speed with logging was 49% of the original speed.

5.3 Effect of logging on space usage

Table 3 shows the effect of logging on working set. Memory usage is measured using the peak working set, as reported by the operating system. We normalize the peak working set for logging to the peak working set for the original programs. In some cases, logging has no effect. The xlisp program allocates

Name	LOC	Description	Modifications
compress	2,998	C# version of SPEC95 compress_129 compressing and decompressing large arrays of synthetic data.	Wrap compress/decompress function in try_all
xlisp	8,950	C# version of SPEC95 li_130	Interpet each LISP file in a separate try_all
go	31,860	C# version of SPEC95 go_099	Place entire game play in a try_all
cmp	244	C# version of IMPACT benchmark cmp on two large files	Place entire compare method in try_all
linpack	635	C# version of LINPACK on 1000x1000 matrix	Run benchmark in a try_all
othello	562	C# version of an Othello playing program	Play entire game in a try_all
wc	186	C# version of the UNIX wc utility on a large file	Run benchmark in a try_all
inlining	> 20,000	Bartok inliner (includes cleanup opts)	Wrap the per-method code in try_all
qsort	270	Quicksort array of 4,000,000 pairs of integers	Wrap entire sort in try_all

Table 1: Benchmark programs

Name	CLR (%)	Original (%)	Logging (%)
compress	94	100	28
xlisp	82	100	35
go	92	100	56
cmp	99	100	32
linpack	100	100	24
othello	91	100	100
wc	96	100	40
inlining	125	100	49
qsort	96	100	50

Table 2: Effect of logging on normalized program speed

Name	Logging (% of original peak working set)
compress	491
xlisp	1050
go	137
cmp	280
linpack	437
othello	101
wc	152
inlining	104
qsort	252

Table 3: Effect of logging on normalized peak working set

a large amount of data that is retained during the try_all block. The high number of heap locations written in compress and linpack programs lead to a large amount of logging data.

5.4 Code size

Table 3 shows the effect of try_all on code size. We measure the size of the text segment at the object file level. We see that code size increases by 1% to 47%. This could be improved by doing a better static analysis to determine what virtual methods are callable from a try_all.

6 Related work

There has been a widespread adoption of exception handling as a mechanism for dealing with errors in modern programming environments. There has been some work done attempting to automate this task. Additionally, some work has been done integrating the concept of a transaction, familiar from databases, into programming languages.

6.1 Exception handling

Exception handling in most widely-used programming languages dates back to several 1970s papers. Goodenough [5] discusses a wide range of existing and possible semantics for exception handling. He proposes structured exception handling, which unlike prior constructs associates handlers lexically with the statements raising the exception. Structured exception handling also allows the handler of an exception to decide either to resume execution after the statement that faulted (resumption semantics) or to terminate execution of the statement that failed. The CLU language adopted structured exception handling [1]. However, it dropped the notion of resumption semantics because it makes it difficult to prove correctness of programs and it inhibits optimization. Most subsequent language designs have followed this decision.

Horning *et al* [10] propose basic recovery blocks. Recovery blocks try to ensure that a condition is met by executing a sequence of actions until the condition becomes true. If an action does not ensure the condition holds, its changes to state are rolled back. They have the syntax: **ensure** *condition* **by** *action 1* **else**

Name	Logging (% of original code size)
compress	123
xlisp	143
go	147
cmp	135
linpack	119
othello	118
wc	131
inlining	134
qsort	101

Table 4: Effect of try_all on code size

action 2 **else** *action 3* ... Checkpointing is done by keeping a cache of old values of memory locations.

Melliar-Smith and Randell [14] describe flaws with structured exception handling: a programmer has to anticipate all the points at which an exception may occur and the types of exceptions that may occur. They argue that it will be difficult for a programmer to properly anticipate exception points because they can occur at so many places. They advocate combining basic recovery blocks with structured exception handling. Cristian also argues for combining exception handling with some form of state rollback [2].

Weimer and Necula [15] analyze common mistakes in implementing error handlers. They create tools to automate this analysis, attempting to find code paths that may not correctly release resources. They assume that programmers reliably insert error handling code where appropriate, and attempt to ensure that this code correctly discharges its responsibilities. They find that all of the programs they analyzed contained at least some paths which did not use resource safely. To support automating recovery, they introduce compensation stacks Compensation stacks allow a programmer to specify how to compensate for an action, and guarantee that the compensation will take place. Note that (unlike registered IUndo objects) compensations are always executed, even if the code runs without error.

Fetzer *et al* [3] classify methods as either *failure atomic* or *failure nonatomic*. They attempt to automatically find methods that do not properly restore state and mask the problem by creating a wrapper that automatically restores state rather then attempt to find and automatically fix buggy code as they do.

6.2 Programming language support for transactions

Some programming languages provide support for transactions. Transactions combine several features: concurrency control to ensure consistent access to data, state rollback, and committing data to persistent storage. These languages have been too heavyweight for many uses. In part, this is because it is difficult to provide lightweight concurrency control for data access that provably prevents data races. In contrast to transactions, we provide a lightweight mechanism for state rollback that eliminates errors in sequential code.

Argus [13] introduces the notion of an atomic regions into programming languages. An atomic region of code is code that is executed in an all-or-nothing fashion. Because of the cost of ensuring atomicity, only same objects are made atomic. Consistency is implemented by acquiring a reader or writer lock for an object before accessing or modifying any fields of the object. Rollback is provided by making a copy of an object before modifying it.

Haines *et al* [6] add transactions to Standard ML. They separate transactions into 4 components, one of which is an undo function. The undo function is a higher-order function that takes another function f as an argument and executes f. If f throws an Abort exception or has an uncaught exception, the undo method undoes all changes to state. Their undo function is not integrated with exception handling. Their implementation, like ours, uses logging of all changes to state.

More recently, Harris and Fraser [9] add atomic blocks to Java. An atomic block is essentially treated as a transaction. This is implemented on top of a software transactional memory (STM) system they develop. The STM uses a form of logging to keep track of old data in case the transaction must be aborted. It also needs to keep track of other data associated with each object, such as the current owner. In contrast to our implementation, the STM imposes an additional overhead on both reads and writes. These atomic blocks are used to allow lockless synchronization, as they can be reverted and restarted if a conflicting write is detected. Harris separately discusses [8, 7] possible extensions to better handle external actions. Atomic blocks guarantee more to the programmer then try_all blocks do, and consequently are more costly.

References

[1] Russell P. Atkinson, Barbara H. Liskov, and Robert W. Schiefler. Aspects of implementing CLU. ACM, 1978.

[2] Flaviu Cristian. A recovery mechanism for modular software. In *Proceedings of the 4th Inter-*

national Conference on Software Engineering, pages 42–50, Piscataway, NJ, USA, 1979. IEEE Press.

[3] Christof Fetzer, Pascal Felber, and Karin Hogstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Trans. Softw. Eng.*, 30(8):547–560, 2004.

[4] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, March 2000.

[5] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[6] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Trans. Program. Lang. Syst.*, 16(6):1719–1736, 1994.

[7] Tim Harris. Design choices for language-based transactions. Technical report, Computer Lab Technical Report 572, August 2003.

[8] Tim Harris. Exceptions and side-effects in atomic blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP 2004)*, July 2004.

[9] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[10] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Proceedings of the International Symposium on Operating Systems: Theoretical and Practical Aspects*, volume 16 of *Lecture Notes in Computer Science*, pages 171–187, Rocquencourt, France, April 1974. Springer Verlag.

[11] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal, and Wen-mei W. Hwu. Optimizing NET compilers for improved Java performance. *Computer*, 30(6):67–75, June 1997.

[12] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen mei W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *IEEE Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.

[13] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.*, 5(3):381–404, 1983.

[14] P.M. Melliar-Smith and B. Randell. Software reliability: The role of programmed exception handling. In *Proceedings of the ACM Conference on Language Design For Reliable Software*, volume 12 of *ACM SIGPLAN Notices*, pages 95–100, Raleigh, NC, 1977. ACM Press.

[15] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, October 2004.