C++ Exception Handling for IA-64

Christophe de Dinechin
Hewlett-Packard IA-64 Foundation Lab
ddd@cup.hp.com

Abstract

The C++ programming language offers a feature known as exception handling, which is used, for instance, to report error conditions. This technique can result in more robust software. On the other hand, it generally has a highly negative performance impact, even when exceptions are not actually thrown. This impact is especially important on an architecture such as the HP/Intel IA-64 processor, which is very sensitive to compiler optimizations. Hewlett-Packard implemented exception handling for IA-64 in a way that leaves the door open for optimizations, even in the presence of exceptions.

1. Overview of C++ Exception Handling

Most software has to deal with exceptional conditions, such as insufficient resources, missing file or invalid user input. In C, such a condition is typically reported using special return codes from functions. For instance, the ubiquitous `malloc` function indicates an out-of-memory situation by returning a NULL pointer. Typical C code would test this situation as follows:

```
void *ptr = malloc(1000000);
if (ptr == NULL)
   fprintf(stderr, "Sorry, out of memory\n");
```

C++ exceptions are a better way to report such a condition. A C++ function that detects an exceptional situation can *throw an exception*, which can be caught by any of the calling functions using an exception handler. For instance, the previous code could be written in a C++ program as follows (the error test is in bold):

```
struct OutOfMemory {};
struct Resource {
   Resource();    // Ctor allocates resource
   ~Resource();   // Dtor frees resource
};

int foo(int size) {
   void *ptr = malloc(size);
   if (ptr == 0)
      throw OutOfMemory();
   /* Do something else */
}

int bar(int elements) {
   Resource object;
   int result = foo(2 * elements);
   /* Do something else */
}

int main() {
   int i;
   try {
      for (i = 0; i < 100; i++)
         bar(i);
   } catch (OutOfMemory) {
      /* Report out-of-memory condition*/
      cerr << "Out of memory for i="
         << i << endl;
   } catch (...) {
      /* Report other problems. */
   }
}
```

If the anomalous situation is detected (in this case, `malloc()` returning zero), the function can report it by throwing an exception. Note that this would not even be necessary had the memory allocation been done the C++ way, since the C++ allocation operators normally report an out-of-memory condition by throwing a standard exception (`std::bad_alloc`). Compared to the C error reporting method, the benefits are multiple:

- The exceptional situation is identified by a specific entity, an exception, rather than by a special return code.
- There is no need for intermediate functions, such as `bar`, to do anything to deal with the exceptions.
- In particular, objects with destructors such as `object` are properly destroyed when the block containing them is exited, whether normally or because of an exception. This makes resource management safer.
- The exception handling code (in `main`) is easily identified as such, and separate from normal processing. A `catch` block catching the exception type `OutOfMemory` is called an exception handler for `OutOfMemory` exceptions.

Throwing an exception involves *unwinding* the call stack until an exception handler is found. This process is made more complex in the presence of C++ automatic

objects, since these objects may have destructors. In that case, destructors have to be called as the stack is being unwound.

2. Performance Impact of Various Solutions

Since exceptions occur infrequently, the performance of exception handling code is normally not critical. In addition, developers can easly control how their application uses exceptions, and avoid exceptions in performance-critical code.

On the other hand, implementations of exception handling generally have a negative performance impact on the code that may throw an exception (the code inside a `try` block), whether this code actually ever throws an exception or not. Ideally, the code inside the try block should not be different than the same code outside a try block. In practice, however, the presence of a try block, or even the presence of an "exceptions are enabled" option in general slows down the code and increases its size. The reasons are multiple and complex. We try to address some of them below.

The performance of an exception-handling solution is therefore measured by its impact on the non-exceptional code when no exception is thrown; it should try to minimize the degradation of code speed and size for this "normal" code.

2.1 Portable Exception Handling with setjmp

The first implementations of C++ exception handling used a mechanism based on the standard C `setjmp` and `longjmp` functions. The `setmp` function saves an execution context in a `jmp_buf` structure. The `lonjmp` function can later be used to perform a "non-local goto", transferring control to the place where `setjmp` was originally called, as long as the function containing the `setjmp` never returned.

In "portable" exception handling, a try block is replaced with a `setjmp` call, and throwing an exception is replaced by a `longjmp`. A linked list of `jmp_buf` buffers will represent the dynamic list of enclosing try blocks. This same technique had been used routinely in C and C++ to simulate exceptions before exceptions became available as a standard language feature.

The major difficulty with this approach is to correctly destroy automatic objects (such as the `Resource` object in our example). This is typically solved by creating a linked list of the objects to be destroyed as you create them. This approach is relatively simple, and it works with a C++ compiler that generates C code, such as the original Cfront from AT&T — this is the reason it is called "portable". This scheme has been used quite widely, in particular by the Cfront-based C++ compiler from Hewlett-Packard [1].

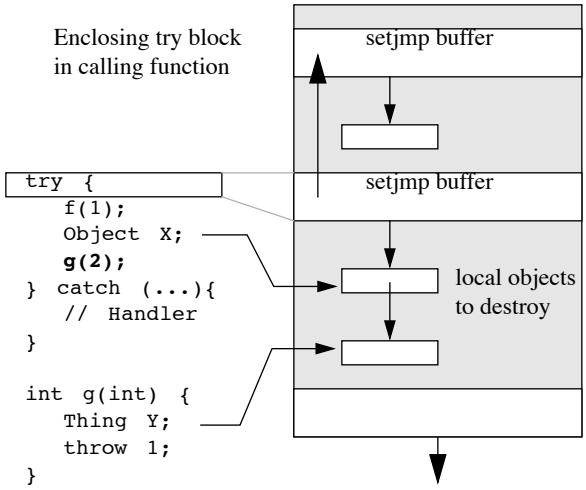

Figure 1. Setjmp based exception handling

On the other hand, the performance drawbacks are significant.

- The `setjmp` function must be called at the beginning of every try block, and the list of `jmp_buf` must be maintained.
- A linked list of objects on the stack must be maintained at all times, and kept in a consistent state with respect to the list of `jmp_buf`.
- All variables that are stored in registers and that are declared outside the try block have to be restored to their initial value when `longjmp` is invoked[1]. For instance, the value of `i` in the `catch` block in `main` must be the same value as when `bar` was called. This can be achieved either by spilling all variables to memory before calling `setjmp`, or by having `setjmp` itself save all registers. Both options are expensive on architectures with large register files such as RISC processors.

This impact exists even if no exception is ever thrown, since the calls to `setjmp` and the management of the

1. Typically, `setjmp` will not save all registers in the `jmp_buf` it is given as an argument. This is why the documentation for these routines generally states something like: "*Upon the return from a setjmp() call caused by a longjmp(), the values of any non-static local variables belonging to the routine from which setjmp() was called are undefined. Code which depends on such values is not guaranteed to be portable.*" (from the HP-UX 10.20 man page for `setjmp`).

object stack have to be done each time a try block is entered or exited.

2.2 Table-Driven Exception Handling

Another implementation of C++ exception handling uses tables generated by the compiler along with the machine code. When an exception is thrown, the C++ runtime library uses the tables to perform the appropriate actions. Conceptually, this process works as follows:

- A first table is used to map the value of the program counter (PC) at the point where the exception is thrown to an action table.
- The action table is used to perform the various operations required for exception processing, such as invoking the destructors, adjusting the stack, or matching the exception type to the address of an exception handler. For example, there will be an action kind to indicate "call the destructor for object on the stack at stack offset N," which will be used to invoke the destructor of the `Resource` object.
- Once an exception handler (a `catch` block corresponding to the type of the exception being thrown) is found, a new PC value is computed from the tables that corresponds to this handler, and control is transferred to the handler.

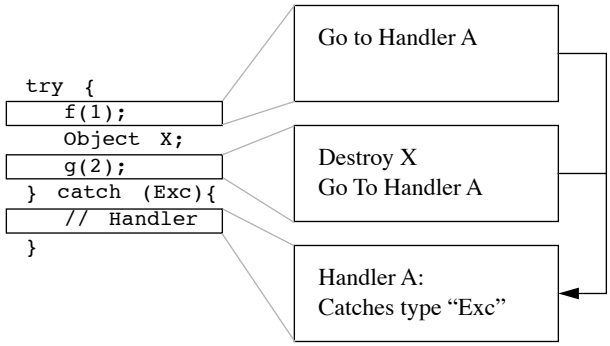

Figure 2: Table Based Exception Handling

This approach is significantly more efficient than the previous one. There is no longer the systematic cost of a `setjmp` function call for every `try` block. Similarly, the cost of maintaining linked lists even when exceptions are not thrown is also eliminated. Therefore, many C++ compilers switched to a table-driven exception-handling mechanism. The Hewlett-Packard aC++ compiler for PA-RISC uses this technique.

On the other hand, there are still negative effects from a performance point of view:

- The runtime needs to be able to restore all variables that are declared outside the try block to their correct value before entering a catch block (for instance `i` in the `catch` block of `main` in the example above.) The impact of this on performance is quite subtle and has multiple aspects, which are discussed below.
- All objects that have destructors must have their address stored in a table. Therefore, they must reside in memory, and their address is implicitly exposed.
- All automatic objects that have their address exposed have to be committed to memory before any call. In practice, this is not often a significant constraint, since a C++ object's address is exposed through the `this` pointer after any member function call (including the constructor.) On the other hand, this may impact the most performance-critical objects, whose member functions are all inlined. These objects could otherwise be promoted to registers.
- The tables themselves have to encode a lot of possible actions, including call to destructors. Therefore, they use a significant amount of space.
- Since tables refer to code, reorganizing the code implies reorganizing the tables accordingly. While this does not preclude optimizations on a theoretical ground, it practically disqualifies any existing optimizer that does not specifically know about C++ and the exception-handling tables.

2.3 Extension of Variable Lifetime

The following code illustrates one problem related to preserving the value of local variables in the presence of exception handling:

```
void f() {
   int x = 0;
   x = f1(x);
   f2(x);
}
```

A smart compiler can discover that the only use of the initial value of `x` is for calling `f1`, at which point it is known to the compiler that the value is zero. Then, `x` gets immediately overwritten with a new unknown value. Therefore, the compiler can legally rewrite the code as follows:

```
void f() {
   int x = f1(0);
   f2(x);
}
```

However, if the above code were to be placed in a try block, this transformation would no longer be valid. For instance, the value of `x` could be used in the catch block:

```
void f() {
   int x = 0;
   try {
      x = f1(x);
      f2(x);
```

```
    } catch (...) {
       cout << "The value of x is " << x;
    }
}
```

This phenomenon extends the lifetime of a variable, and therefore puts additional pressure on the register allocator. It also makes the control flow much more complex, by creating additional potential control-flow arcs between any call and each of the catch clauses. As a result, register usage will tend to be much more conservative within a try block than outside of it. On the other hand, these effects occurs only in the presence of a try block: destructors, for instance, cannot access local variables whose addresses have not been exposed.

2.4 Register Selection Constraints

Another slightly different problem can be shown in the following code:

```
int f(int x) {
   x = f1(x);
   return f2(x);
}
```

A smart compiler can notice that the initial value of `x` becomes "dead" right after the call to `f1`, since the result overwrites the previous value of `x`. The same thing happens with the second value of x, which lives only until the call to `f2`. So the compiler can rewrite the code as follows:

```
int f(int x) {
   int x2 = f1(x);     // and discard x
   int x3 = f2(x2);    // and discard x2
   return x3;
}
```

This alternative leaves much more freedom in terms of register allocation, since now different registers (or memory locations) can be allocated for `x`, `x2` and `x3`. In particular, this means that the value of the register used needs not be preserved accross the function calls. But this freedom does not exist if the above code is enclosed within a try block. In that case, the catch clause may access variable `x`, and therefore all `x` values have to live in the same register. Again, this problem occurs only in the presence of a try block.

2.5 Control-Flow Complexity

In the presence of a try block, the control flow becomes much more complex, since an implicit "goto" exists between any function call or `throw` statement in the try block and each `catch` block. Another implicit "goto" exists between the end of each catch block and the end of the function. This can impact optimizations on the following code:

```
for (i = 0; i < 1000; i++)
   x = f(i) * 3 + 1;
```

Outside of a try block, the code in question has a rather well known behavior, so if `x` is not address exposed and can therefore not be visible inside `f`, the compiler can predict that the value of x on exit from the loop will be the value computed at the last iteration. In other words, it can postpone the multiplication and addition until after the loop. In real code, the computation would probably be implicit, for instance taking the address of a struct element (an addition), or of an array element (multiplication by the element size).

Optimizing away the computation can't be done if there is a try block surrounding the code, since in that case any of the `catch` blocks can read the value of `x`.

2.6 Memory Access Order

Memory accesses are more strictly ordered in the presence of exceptions. This effect is quite significant, because it occurs even without the presence of a `try` block. Consider the following code:

```
struct Object { float x, y; ~Object(); };
Object object;
for (int i = 1; i < 1000; i++) {
   object.x += f(i);
   object.y += g(i);
}
```

In this code, the compiler can identify that for a normal iteration of the loop, memory accesses can be avoided, and replace the loop code with something like:

```
register float tmp_x = object.x;
register float tmp_y = object.y;
for (int i = 0; i < 1000; i++) {
   tmp_x += f(i);
   tmp_y += g(i);
}
object.x = tmp_x;
object.y = tmp_y;
```

Of course, if `f` or `g` can throw exceptions, then the destructor has to see the correct value of the object, and the invocation of the destructor can occur at any time. So the compiler must generate code that writes the value of the object to memory in the original source order with respect to function calls.

In practice, this last effect and its variants tends to be the most significant, since it affects memory accesses, which are expensive on today's microprocessors, and it occurs whenever exceptions are enabled, regardless of whether there are exception-related constructs in the code.

3. IA-64 Exception Handling

The various problems listed previously can be classified in one of the two following categories:

- Cost of saving and restoring registers
- Ordering constraints due to additional arcs in the control graph

The first problem is addressed in a rather original way by a feature of the IA-64 architecture called the "*Register Stack Engine*" (RSE) [3]. The RSE defines a standard way to save and restore registers when entering and exiting functions which is not directly under program control. As a result there is no real need to explicitly save registers, yet there is a way for the runtime to restore them to their original value.

The second problem is addressed in our implementation of C++ exception handling by allowing the non-exceptional path to be optimized, as long as *compensation code* is placed along the exceptional paths to restore program state before executing the exception handlers to what it would have been if the optimization had not taken place. The place where such compensation code is added is called a *landing pad*, and serves as an alternate return path for each call.

3.1 Register Stack Engine and Unwind Table

The IA-64 architecture features numerous registers [2]. The integer register set is partitioned into 32 fixed registers and 96 "*stacked*" registers. The stacked registers are automatically saved on a special stack, using free cycles in the load-store unit whenever possible.

Registers are typically not stored to memory immediately on function entry. Instead, stacked registers are renamed so that the first stacked register for the current procedure is always called `r32`. Dirty (non-saved) registers are pushed on the stack when calls are made, while previously saved registers are popped from the stack and restored when calls return. The processor tries to save as many registers for future calls and restore as many registers for future returns as free memory bandwidth and free registers allow. This technique maximizes the chances that a function call or function return can be performed without memory accesses to save or restore registers.

The Register Stack Engine which performs these operations is itself a very complex topic that would require an article in its own right [3]. For C++ exception handling, however, the key feature of the RSE is the way it transparently saves and restores stacked registers "in the background", and does so at locations on the stack that the runtime can compute.

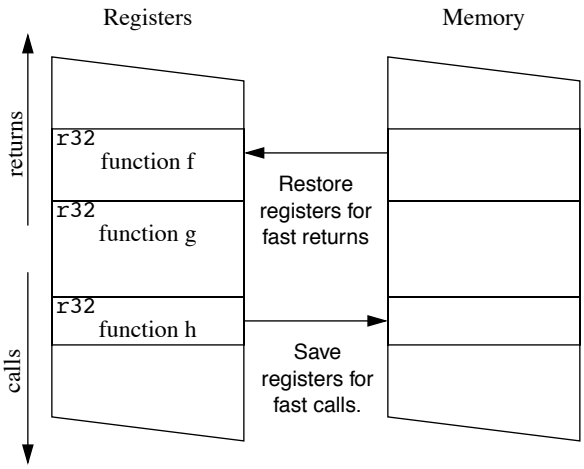

Figure 3: IA-64 Register Stack Engine

When an exception is thrown, the runtime forces the RSE to flush all stacked registers onto the stack. It can then manipulate them as needed, and later let the RSE restore them as the stack is unwound.

Only stacked registers are saved this way. The IA-64 runtime architecture [4] indicates that non-stacked registers are saved in stacked registers. Floating-point registers are saved using more traditional mechanism. The information indicating where each particular register is being saved is stored in separate tables, called *unwind tables* [5].

Together, the unwind tables, stack unwinding routines [6] and the RSE help restore register values to the exact same state they were in any given function, without the runtime cost of saving them "manually" in each function.

3.2 Exception Handling Tables

Restoring registers to their previous state is necessary, but not sufficient for throwing a C++ exception. The C++ runtime also needs to call the destructors, to find the appropriate exception handler, and to transfer control to this exception handler.

The information required to do this is found in *exception handling tables*. These tables are C++ specific. They contain information to map call sites to the landing pads. Each landing pad will process any exception thrown from the corresponding call site, and serve as an alternate return point for this call. The table also contains information regarding which exceptions the landing pad can process and catch, and records exception specifications if any.

The reason the table maps call sites is that our C++ implementation only throws from a call site. The `throw` keyword itself is implemented by calling a runtime routine. Errors such as division by zero or invalid memory accesses do not need to throw exceptions in C++ [7] (they are

"undefined behavior"). Practically all implementations use alternative mechanisms such as Unix signals[1].

Therefore, from a machine language point of view, the only places that can throw are call instructions. If the program counter is within the range of a given subroutine but no call site matches, the runtime will call the `terminate()` function, as specified in C++.

An interesting implication: this mechanism does not allow C++ exceptions to be thrown out of a Unix signal handler, something that the ISO C++ Standard specifically discourages [7] (Clause 18.7, paragraph 5, restricts signal handlers to what can also be written in plain C). For instance, if a memory access instruction causes a signal, and if the signal handler throws an exception, the program counter of the function containing the memory access will not be on a call instruction, and `terminate()` will be called.

3.3 Landing Pads

The runtime transfers control to a landing pad whenever an exception is thrown from a given call site. The landing pad will contain code in the following order:

- *Compensation code*, restoring program state to what it would be if optimizations had not been done in the main control flow.
- *Destructor invocation* to destroy any local object that needs to be destroyed.
- *Exception switch* to select which catch handler, if any, to jump to. An appropriate switch value is computed by the runtime from the C++ exceptions table, and placed in a temporary register.
- A *landing pad exit*, which either returns to a catch block, and from there to the main control flow, or resumes unwinding if no appropriate exception handler is found in this subroutine.

The same mechanism can deal with all kind of destructors (inlined or not, array destructors, …), which all had to be special table entries in a table-driven exception handling runtime. A catch-all exception handler (`catch(...)`) is simply a default exit in the exception switch.

Together, the landing pads form a funnel where the compensation code can be somewhat different for each call site, while destructor code is shared for sections of code between declarations, and the exception switch and landing pad exit are shared for all code within the same try block.

1. One notable exception is the Win32 platform, where system exceptions and C++ exceptions interact.

3.4 Compensation Code

It is relatively easy for the compiler to generate compensation code for any of the operations listed in previous sections:

- If the variable is allocated to a different register in different sections of code, the landing pad can simply copy that register to the target register which represents the variable in the exception handler.
- A value known to be constant which has been replaced with the constant value can be loaded into the appropriate register by the landing pad, for use by the user code in the exception handler.
- Pending memory operations that have been delayed in the main flow of control can simply be executed in the landing pad should an exception be thrown.

Compensation code therefore allows the compiler to utilize any of the optimizations that were prevented by simpler table-driven techniques.

Since the IA-64 architecture is very sensitive to optimizations, the ability to insert compensation code alone is a compelling reason for selecting a landing pad based approach. On other architectures, the benefit of landing pads may not be high enough to compensate for the code size penalty compared to other table-driven techniques.

3.5 Placing Landing Pad in "Cold" Code

Landing pad code is not used except when an exception is thrown. If the landing pad code is simply placed at the end of the code for each function, the useful code becomes interspersed with blocks of little-used code. This can affect paging and caching performance, since the exception handler code will occupy space in the various memory caches and in the virtual memory active set.

For this reason, landing pad code can be placed in a different code section. This code can be placed at link time arbitrarily far from normal "hot" code. The hot code can then be kept contiguous and makes better use of the cache and virtual memory pages.

Even when it does not actually use cache lines or virtual memory active pages, landing pad wastes space on disk when it is not used. Since it is in general infrequently used, landing pad code can therefore be space optimized rather than speed optimized.

3.6 Compressing Tables

The overhead of exception handling includes the exception handling tables. These tables are not used except when an exception is thrown. Just like landing pad code, they can be placed arbitrarily far from the code so as to

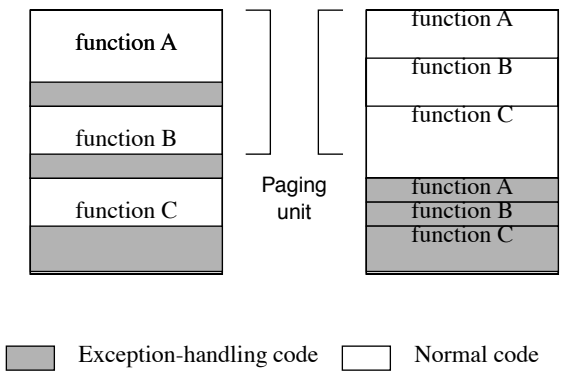

Figure 4: Separating Hot and Cold Code

minimize impact on caching and virtual memory. On the other hand, they still use valuable disk space in the executable image.

The exception handling tables in the Hewlett-Packard aC++ compiler use a compression scheme known as "LEB128". This encoding uses less space for small values: 1 byte for any value less than 128, 2 bytes for any value less than 16384 and so on. So the tables contain relative offsets that are often small. For instance, the first call site address is encoded as the number of 16-byte instruction bundles from the start of the function, and later call sites are relative to the previous call site in the table. This helps keep the offsets small enough to fit in one or (rarely) two bytes.

Compressing the tables has a slight negative effect when an exception is actually thrown, since the table contents need to be decoded. In that infrequently executed case, we traded speed for space.

3.7 Known Functions That Can't Throw

When exceptions are enabled, there is a landing pad and table space overhead for each call site. This overhead can be avoided for specific functions that are known not to throw. These functions typically include:

- C++ runtime library functions called implicitly by the compiler.
- Functions of the C library, since they are known not to throw exceptions.
- Functions that are marked as not throwing exceptions through an empty exception specification. Exception specifications are verified inside the function, not at the call site.

If such a call ever throws, the C++ runtime will call `terminate()`.

3.8 Remaining Negative Effects

Even with landing pads, exception handling still has a cost. The space overhead of enabling exception handling includes the code for the landing pads, exception switches, destructor calls and catch handlers, as well as the space for all the exception handling tables. This remains significant in terms of memory usage, even though the performance impact of this additional memory can be kept low by carefully segregating hot and cold memory.

However, performance itself can remain affected by a variety of factors:

- Additional control flow arcs between the main code call sites and the various exception handlers and destructor calls make the control flow graph much more complex.
- One result is to prevent some otherwise valid code motion, when the code motion cannot be correctly compensated for in the landing pad, or when the cost of compensating would be too high.
- Another effect is to effectively lower the amount of optimization that can be done on a given piece of code in a given amount of time. Since compilers also have a compile-time performance constraint, they may "bail out" if optimization would take too long. This will happen earlier in the presence of exception handling.
- Another instance of resource limitation occurs on optimizations that copy or duplicate code, such as inlining. These optimizations typically have a "budget", and this budget gets exhausted much more rapidly in the presence of exceptions, since duplicating the main body of code generally means duplicating the exception-handling data and code as well.
- A final problem evoked above remains unsolved: any optimizer that performs on the code has to know about exception handling tables and how to reorder them. With limited engineering resources, some specific optimizations may purposedly be disabled in the presence of exceptions.

4. Results

The following tables records timing and size measurements done on various benchmarks. These have been run on a performance simulator for the next generation IA-64 processor, and remain to be validated on real hardware. For comparison purposes, similar meaurements have been done on current generation PA-RISC processors. Only relative results are shown, since absolute SPEC results for IA-64 have not been published yet.

The values measure the performance penalty when enabling exception handling. For speed, the penalty is the additional number of cycles in the simulator. For memory, it is the additional size of text and initialized data, as reported by the `size` command.

These benchmarks contain a mix of C and C++ application code, but they often do not rely very much on C++ local objects or exception handling. Therefore, they represent a worst case but not very uncommon scenario where exception handling is not used and therefore you don't want to pay for it. Some of the benchmarks were originally written in C and have been modified to be compilable with a C++ compiler. Table 1 records size and speed penalties. In general, measurements were taken at the maximum optimization level, except for the last two rows where the optimization level was +O1.

Table 1. Speed and Size Penalty

	IA speed penalty	IA size penalty	PA speed penalty	PA size penalty
099.go	3.73%	-9.58%	15.6%	0.21%
129.compress	-5.66%	<0.01%	2.00%	< 0.01%
130.li	-6.56%	-14.19%	11.15%	1.24%
132.ijpeg	-0.21%	-0.48%	-0.49%	0.06%
134.perl	-1.43%	-18.49%	0.81%	1.02%
147.vortex	N/A	-8.08%	0.66%	-0.04%
Raytracer (+O1)	-1.6%	16.8%	0.3%	6.13%
C++ Library (+O1)	N/A	0.2%	N/A	0.2%

Suprisingly, in some benchmarks, enabling exception handling actually yields better performance. This has to be taken with a grain of salt. At this point, it is quite difficult to do accurate IA-64 measurements, whether on real hardware or on a simulator. For instance, simulator results are sampled, and the sampling noise alone can account for a few percents of variation either way. Similarly, optimizer "luck" in scheduling instructions can also introduce unpredictable variations. Therefore, the "noise level" of these measurements is quite high. You should not expect code to become faster because of exception handling.

The size aspect is even more surprising, as shown on Table 2 below.

Table 2. Size penalty for various optimizations

	+O1	+O2	+O3
099.go	9.41%	9.07%	-9.58%
124.m88ksim	18.83%	13.02%	-11.30%
129.compress	0.01%	<0.01%	<0.01%
130.li	35.73%	24.87%	-14.19%
132.ijpeg	0.72%	0.42%	-0.48%
134.perl	28.80%	28.57%	-18.49%
147.vortex	37.48%	30.57%	-8.08%

Processing exceptions requires additional code. The effect above on the IA-64 compiler shows only for the maximum optimization level (+O3), and may actually indicate a problem with the tested compiler. At lower optimization levels, the PA-RISC compiler consistently produces executables of the same size with or without +noeh. The IA-64 compiler produces executables that are significantly larger with exception handling enabled, which is what one would expect.

5. Analysis

Overall, the objective of minimizing the negative runtime performance impact of exception handling at high optimization levels is achieved. This constrasts with PA-RISC, where penalties as high as 15% are observed (and in practice 10% is not uncommon). It remains to be seen if this conclusion remains valid as more aggressive optimizations are added to the IA-64 compiler.

The size penalty on IA-64 tends to be higher, at least at optimization levels used during application development. This is due largely to cleanup code, which takes more space than the same information stored in PA-RISC action tables. As usual, there was a space versus time trade off, and this technology definitely favored speed.

The added exception handling code is normally infrequently executed. Modern operating systems do not load code into memory until it is about to be executed. So most of the time, the additional code just consumes disk space, without necessarily increasing the memory footprint of the application. Disk space gets cheaper all the time, so the trade off was a reasonable one.

This code size penalty may be reduced somewhat in the production compilers by a change in the C++ Application Binary Interface (ABI) [6] that is not implemented in the tested compiler. This change reduces the size of a minimal landing pad from 32 bytes down to 16.

The reason for the size reduction at maximum optimization has not been investigated yet. It may indicate a problem with the compiler, such as an optimization being accidentally turned off when exceptions are disabled. Exception handling may also prevent some code-expanding transformations that look less profitable, such as inlining and loop unrolling. It is unclear if these results will persist with a production compiler.

In general, keep in mind that all measurements above were made with a largely prototype compiler, and in a simulator. As our understanding of optimization techniques specifically targetting the IA-64 architecture improves, the results may change significantly.

6. Conclusion

Landing pads offer an interesting alternative to more traditional implementations of C++ Exception Handling. They leave more optimization freedom to the compiler. Many aggressive optimizations can now be performed equally well even in the presence of exception handling code, making applications that require exception handling faster.

One the design objectives of C++ is that you don't pay for features that you don't use. This objective was not met with many exception handling implementations. The IA-64 implementation presented here is one more little step towards that goal.

This design has been shared by Hewlett-Packard with other Unix vendors, and should hopefully become available on a variety of IA-64 platforms as part of the effort towards a common C++ Application Binary Interface for IA-64 on Unix [6].

7. References

[1] *A Portable Implementation of C++ Exception Handling* D. Cameron, P. Faust, D. Lenkov and M. Mehta, Proc. USENIX C++ Conference, August 1992.

[2] *IA-64 Instruction Set Architecture Guide* Revision 1.0, Intel Corporation / Hewlett-Packard Company
http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/index.html

[3] *IA-64 Register Stack Engine* Chapter 9 of the "*IA-64 Instruction Set Architecture Guide*" [2]
http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/rse.html

[4] *IA-64 Software Conventions and Runtime Architecture* Vers. 1.0, Hewlett-Packard Company
http://devresource.hp.com/devresource/Docs/Refs/IA64CVRuntime.pdf

[5] *Stack Unwinding and Exception Handling* Chapter 11 of "*IA-64 Software Conventions and Runtime Architecture*" [4].

[6] *C++ ABI for IA-64: Exception Handling* Working document, C++ ABI Committee
http://reality.sgi.com/dehnert_engr/cxx/abi-eh.html

[7] *ISO 14882: C++ Programming Language*
http://www.iso.ch

Thanks to M. Brown, D. Cameron, C. Coutant, D. Handly, E. Gornish, D. Gross, R. Ju, and D. Vandevoorde for their contributions to this paper or the techniques it describes.