Optimizing Away C++ Exception Handling

Jonathan L. Schilling

SCO, Inc.
430 Mountain Avenue, Murray Hill, NJ 07974
jls@sco.com

Abstract

A high performance implementation of C++ exception handling is crucial, because exception handling overhead is distributed across all code. The commonly-used table-driven approach to implementing exception handling can be augmented by an optimization that seeks to identify functions for which (contrary to first appearance) no exception handling tables need be generated at all. This optimization produces modest but useful gains on some existing C++ code, but produces very significant size and speed gains on code that uses empty exception specifications, avoiding otherwise serious performance losses.

Keywords

C++, exception handling, compiler, optimization, benchmarks.

1. Introduction

Moreso than with many language features, a high performance implementation of C++ exception handling (EH) [C++ 97] is critical. This is because exception handling overhead in C++ is distributed: the code generated for all functions potentially needs to contain exception handling information, even when no explicit EH constructs are used in a function.

Most high-performance EH implementations use the commonly known *table-driven* approach. The goal of this approach is to present zero execution-time overhead, until and unless an exception is thrown. In doing so, the speed of actually handling an exception is sacrificed to whatever degree necessary; this is in accordance with the general design goals of exception handling in C++ [Stroustrup 93] and other languages [Chase 94].

The table-driven approach consists of building logically read-only, static tables at compile and link time that relate ranges of instruction counter values to the dynamic state of the program regarding exception handling. Then, if an exception is thrown, the C++ runtime system looks up the current instruction counter in the tables and performs whatever semantic actions are necessary: give control to a catch handler, invoke the destructor for a local object, detect that an exception specification has been violated, unwind the stack, etc.

Similar table-driven techniques have been used to implement exception handling in other languages, and the technique was envisioned as the basis for high performance implementations when C++ exception handling was first designed [Koenig 90]. A good general illustration of the technique as it applies to C++ is given in [Lajoie 94]; specific implementations vary in the details. A more detailed discussion of the technique for a C++-like language is given in [Christensen 95]. The major alternative approach, *dynamic registration*, is described in detail in [Cameron 92]. It involves generating code that updates run-time data structures about the state of the program regarding exception handling. It is possible to implement exception handling in a portable fashion using this technique (such as when generating C as an intermediate language, or when using a C back end unmodified for C++), and hence it is used in some compilers, but it suffers from significantly slower execution times and is harder to make thread-safe.

Because keeping the distributed overhead low is the primary focus of an exception handling implementation, this paper does not address the speed at which exceptions are thrown or caught, nor for the most part the overhead of establishing try blocks.

2. Overview of SCO C++ Implementation

This paper discusses consequences of the implementation of exception handling in the C++ compiler in the SCO UnixWare/OpenServer Development Kit (UDK)®, which runs on the SCO OpenServer® and UnixWare® operating systems for the Intel x86 architecture.

The SCO implementation of exception handling uses the table-driven approach. It is beyond the scope of this paper to fully describe the implementation, such as the layout of the different kinds of EH table entries, but some particular characteristics of it are:

- All EH processing is performed by the C++ runtime system based on actions described by the tables. No "code snippets" are inserted into the generated code to handle exceptions, other than local register save and restore code emitted at entry to try blocks and catch handlers.

- The type encodings used in matching thrown types with catch handlers and exception specifications are the same encodings as those used for the C++ runtime type information feature [Stroustrup 94]. These encodings are placed in normal data sections, since runtime type information is typically used during normal program execution.
- All other EH tables contain data that won't be referenced unless and until an exception is thrown. They are placed in new, special data sections in the generated code, named `.eh_ranges`, `.eh_reloc`, and `.eh_other`, depending upon the relocation needs of the particular table entries within. Since this data is all grouped together, virtual memory pages containing the data will not need to be referenced and loaded until an exception is thrown.
- All EH processing is thread-safe.

As can be seen, the EH sections are structured partly along the assumption that in a typical normal and successful run of a C++ application, no exceptions will ever be thrown. Whether this is in fact true is debatable; [Stroustrup 97] posits the opposite view. There are probably not enough production applications now using C++ exception handling to be able to make any statistical conclusions, although such a survey might be useful for languages such as Ada that have had exceptions for a long time.

But even if an exception is thrown during execution, this EH section arrangement is beneficial. This is because two of the three section types contain no relocation data. Thus, when the sections appear in dynamic libraries, they are position-independent and are shared across all applications that are accessing those libraries. This benefit was emphasized by [Chase 94] but here is extended to tables governing object cleanup processing as well as tables for locating handlers.

3. Inefficiencies of the Table-Driven Approach

The table-driven technique has three sources of inefficiency: exceptions can take a longer time to process, the tables can take up a lot of space in memory, and in order to maintain the tables' association with instruction ranges, some code optimizations that would otherwise be done may have to be constrained or suppressed entirely.

The first issue is part of the basic trade-off of this design, and the second concern is dealt with as described in the previous section. The most important concern is the third one: indirect overhead due to lost optimization, which imperils the goal of zero execution-time overhead. Lost optimization can come from several sources:

- the necessity to suppress code motion and instruction scheduling that would cause objects' lifetimes to cross over instruction counter markers delimiting them in the tables
- the necessity to teach back ends not to remove catch handlers as unreachable code
- the necessity to keep local destructible objects in memory rather than in registers, so that the EH runtime can find them to destruct them
- an inability to use the most concise forms of stack layouts, due to their not being self-describing in terms of how to walk back from one stack frame to the previous one
- cache misses due to rarely-executed catch handler code sitting in the middle of "main line" code.

All of these problems can be overcome with enough work in the compiler and/or enough expansion of the granularity of the EH tables, and in a better world they would be. In practice, however, C++ back ends (code generators and optimizers) are usually derived from and shared with the back ends of compilers for C and possibly other languages. These back ends were likely not designed with exception handling in mind (unless they have been used for languages such as Ada or Modula-3). How easily they adapt to C++ depends on characteristics of their architecture, but substantially redesigning them may not be feasible on technical, economic, or political grounds.

The SCO C++ compiler suffers to some extent from most of these problems. The venerable "C trees" syntax-based tree structure used for the compiler's intermediate language [Kristol 86] was not set up to express labels that are not branched to. The global optimizer knows of many kinds of data constraints, but does not know of textual code constraints. The optimizers require dummy conditional branches to catch handler code after each call within a try block, to correctly assess the data flow [Cameron 92] [Dean 96]. None of the compilation components has a standard interface for reading and manipulating complex data structures such as the EH tables. Functions with EH tables cannot easily be inlined, due to the tables' contents being determined in the compiler before the decision to inline is made. Only the official, conservative stack layout specified by the application binary interface [ABI 93] may be used, rather than either of the more efficient layouts typically used when optimization is on. Finally, the SCO tool that relocates rarely-used code runs independently of the compilation system and does not have ready access to the EH table information.

While significant effort has been made towards adapting these components to the needs of C++ exception handling, an economical approach is to identify functions where, contrary to first appearance, no EH tables or other indirect inefficiencies need be generated.

This ensures that there will be no indirect loss of optimization at all. Moreover, doing so subscribes to the "early intervention" theory of optimization: the earlier in the compilation process for a high-level language that you can identify an optimization, the better off you are, because it relieves analytical and volume pressure on downstream components.

Lastly, identifying functions that don't need EH tables reduces the total size of those tables. While the tables are already structured so as to limit adverse effects upon system performance when an exception is not thrown, it never hurts to minimize their size. Furthermore, the smaller the tables, the faster the lookup in them when an exception *is* thrown. Finally, the smaller the tables, the less disk space required for C++ executables and dynamic libraries.

4. Optimizing Away Exception Handling

4.1 General principles

There are three basic reasons why a function may need to have EH tables generated for it: it contains a try block, it contains an exception specification, or it contains destructible local objects. In each of these cases, if an exception is thrown out of or through the function, the EH runtime will need tables to tell it the necessary semantic actions to perform.

Furthermore, even if a function does not meet any of the above criteria, it is still possible that an exception will be thrown through it — that is, some function that this function calls may throw an exception. Even though the C++ runtime does not need to do any cleanup actions within this function, it does need to be able to unwind it. As explained in the previous section, this means using a non-optimal stack frame layout on the Intel x86 architecture, and thus represents an indirect inefficiency.

The basic approach to eliminating EH tables and indirect inefficiencies is to identify functions that by their signature, or by examination of their code, are known not to throw an exception; then to propagate this information in a bottom-up way during compilation, to help analyze additional functions. Functions that cannot throw an exception need not have EH generation done for them, even if some or all of the above conditions are true.[1]

How does one know a function cannot throw an exception? If it does not throw any exceptions, and does not call any other functions (or take any other actions) that can throw an exception. (Because C++ exceptions are synchronous, and can only result from C++ throw statements, this analysis is possible.)

This information can come from two sources: information the programmer gives the compiler, i.e. exception specifications, and information the compiler figures out from examining the source as it compiles it.

A simple example of the first is this translation unit:

```
class A {
public:
       A(int) throw();
       ~A() throw();
       int get() throw();
};
int f(int i) {
       A a(22);
       return a.get();
}
```

Because of the empty exception specifications, the compiler can tell that no exception will be thrown from function `f()`.

The exception specification most useful in optimization is the empty one: `A f(const A&) throw();` This says that a call to `f()` cannot result in an exception being thrown back through the caller. If an exception does occur somewhere within `f()` or the functions it transitively calls, it will either have been caught and handled by the time `f()` returns, or control will have passed to the standard library function `unexpected()`.

An example of the second source of information would be:

```
int g(int i) { return i + 1; }

int h(int j) { return g(2 * j) + g(2 - j); }
```

Function `g()` clearly cannot throw an exception, and as a consequence neither can function `h()`. This kind of analysis can be done for all non-virtual function calls. It cannot be done for calls to virtual functions (since the source for the function that gets called may not be visible or even written yet) unless it can be statically determined which function will be called.

Another category of functions known to not throw exceptions are those in the C standard library, most of which is part of the C++ standard library. Except for a couple of cases such as `qsort` and `bsearch` which take function pointers as arguments, these are guaranteed to not throw exceptions [C++ 97].[2]

Of course these two sources of information can be combined. Knowledge of whether a function can throw can be used to optimize the generation of exception specifications themselves:

```
int e(int i, int j) throw() {
       A a(22);
       return h(a.get());
}
```

Normally the empty exception specification of function `e()` would cause the generation of EH tables for it, in order to allow the C++ runtime to detect when an exception specification violation occurs. But because we can analyze the definitions of class `A` and function `h()` and know that no exception can be thrown from them, we do not need to generate EH tables for `e()`. This optimization is of great importance if empty exception specifications are used frequently, as will be shown.

4.2 An algorithm for optimization

Figure 1 shows an algorithm that can be used to scan and mark functions during a one-pass front end compilation. It requires two additional boolean fields in the symbol table entry for functions, *requires_EH_tables* and *may_throw_exception*. Otherwise the algorithm is very inexpensive, since it does not require an extra pass or other elaborate processing.

One somewhat subtle point in Figure 1 is that while scanning the function, *may_throw_exception* refers to whether the function

1. Actually, for internal architectural reasons the SCO compiler does not attempt to eliminate EH tables when a function contains a try block, even if the function cannot throw an exception. Nevertheless this is included in the algorithm presented in Figure 1.

2. Note that `signal` and `atexit` are known not to throw, since the functions passed to them are not invoked at the time of the call. Also, it is necessary to check command options, in case compilation is being done in "freestanding" mode, in which library names may mean something else entirely.

```
as compile each function func in translation unit do
   func.requires_EH_tables := false
   func.may_throw_exception := false

   as compile each declaration/statement in func do
      if see a throw
         func.may_throw_exception := true
      if see a real virtual function call
         func.may_throw_exception := true
      if see a call to a function g
         if function_may_throw_exception(g)
            func.may_throw_exception := true
   end

   if  func.may_throw_exception
   and (func contains a try block
   or func has exception specification
   or func has destructible objects)
      func.requires_EH_tables := true

   if func has null exception specification
      func.may_throw_exception := false
end
```

Figure 1. Algorithm for marking functions in symbol table

can throw an exception *within it*, so that we can know whether to generate EH tables for it. But once the scan is finished, *may_throw_exception* refers to whether the function can throw an exception *out of it*, for analytical use when scanning functions that call this one. A function such as `void f() throw() { throw 1; }` will have the *may_throw_exception* value of its symbol table change from true to false during compilation.

Figure 1 uses the algorithm *function_may_throw_exception*, which is shown in Figure 2. In the back end of the compiler, the new symbol table fields for the function are used to determine what kinds of EH code and table generation are necessary; Figure 3 supplies the decision logic. Figure 4 shows a larger translation unit that requires no EH generation at all; it is used in measurements in the next section.

5. Results

In this section we use some publicly available C++ sources, that in almost all cases do not use exception handling, as benchmarks to measure various effects of the SCO EH implementation. We try to answer the following questions:

1. What is the general overhead of EH in the SCO compiler?
2. How much worse would the overhead be if it were not for the optimization described in Figure 1?
3. If programmers aggressively use empty exception specifications, what will the effect upon performance be, with and without the optimization?

The compilations and executions were done on an Acer Intel Pentium 133 MHz machine with 32MB RAM and 512KB cache,

```
function_may_throw_exception(f) return boolean is
   if f has null exception specification
      -- if f is expression, use its type to determine
      return false
   else if compilation is hosted
   and f is non-pf-passing C std lib function
      return false
   else if f is compiler-generated function
      -- these are known not to throw
      return false
   else if f is not defined
      -- extern or not scanned yet
      return true
   else
      return f.may_throw_exception
end
```

Figure 2. Algorithm to determine if a function can throw

```
as compile each function func in back end do
   if func.requires_EH_tables
      need to generate EH tables for function
      need for RTS to be able to unwind the function
   else if func.may_throw_exception
      don't need to generate EH tables for function
      need for RTS to be able to unwind the function
   else
      don't need to generate EH tables for function
      don't need RTS to be able to unwind the function
end
```

Figure 3. EH code generation based on symbol table values

running SCO UnixWare 2.1.2 and compiled with SCO UDK 7. The compiler implements all EH-related language and library features in the new standard except function try blocks and placement delete. All compilations were done using `CC -O -Kpentium` options. Measurements of code and data size are from the UNIX `size` command. Timings are in seconds and are from the UNIX `timex` command, unless the benchmark captures and prints timings itself, and are taken from the median of three runs in single user state. All programs were linked dynamically, except for a static C math library.

The C++ sources used and where they were found are:

- Figure 4, the example from the previous section (which of course is contrived to show the optimization at its best)
- richards, a small operating system simulator†
- OOPACK, an artificial benchmark to measure OOP overhead‡; containing no destructible objects or EH usage, it

† http://www.cs.ucsb.edu/oocsb/benchmarks/

‡ http://www.kai.com/C_plus_plus/benchmarks/index.html

```
#include <string.h>
class A {
public:
       A(int i) { n = i; count++; }
       ~A() { count--; }
       void set(int i) { n = i; }
       int get() const { return n; }
       int get_count() const;
private:
       int n;
       static int count;
};
int A::count = 0;
int A::get_count() const { return count; }
inline int f(int i) {
       A c(3);
       return c.get() * i;
}
int g(const char* s) { return f(strlen(s)); }
int h(A& a, const char* s) {
       A b(2);
       const int x = g(s);
       A d(x);
       a.set(0);
       return a.get() + b.get() + d.get();
}
int m() {
       A a(1);
       const int y = h(a, "EH");
       return a.get_count() * y;
}
int main() {
       int r;
       for (int n = 1; n <= 1000000; n++)
              r = m();
       return r;
}
```

Figure 4. Example of optimization generating no EH overhead

measures whether EH tables are generated unnecessarily

- the RPI (partial) implementation of the C++ standard library valarray class§ with its example case made into a driver
- deltablue, an incremental dataflow constraint solver†
- The Haney kernels, artificial benchmarks to measure OO overhead in high-performance computing‡
- Newmat, a matrix algebra package§§
- the HP (out of date) implementation of the Standard Template Library with the RPI set of examples* coalesced and made into a driver[3], used because it contains no EH
- ixx, an IDL to C++ parser and stub code generator†
- lcom, an optimizing compiler for an HDL†
- eon, an object-oriented ray-tracing program.‡

In valarray and Newmat loops have been added to the main procedure of the source, to get a measurable running time.

These benchmarks are not a scientifically sampled set of C++ code, nor are the execution times guaranteed to be meaningful (differences up to ±2% are certainly suspect). Both artificial "micro-benchmarks" and larger "real application" sources have been used; some are veterans of previous C++ performance studies (e.g. [Driesen 96]). They do not cover several of the major application areas that C++ is known to be used in (a perennial problem with C++ performance measurement). Nevertheless, it is hoped that the sources are representative enough that the conclusions drawn below are valid.

5.1 Measurement of general EH overhead

Table 1 addresses the first of the above questions. The "EH suppressed" results come from compiling using an undocumented compiler option that disables the exception handling language feature altogether. The "EH enabled" results come from a normal compile with exception handling enabled. Intuitively, the EH suppressed numbers should be better. The "overhead" numbers represent the additional cost of EH enabled compared to EH suppressed: the lower the number, the better.

What do these numbers show? The execution time overhead is small, generally 5% or less. The text size overhead is also fairly small, generally less than 10%.

In some cases the text size overhead is negative, meaning the code is smaller with exceptions enabled than suppressed! This is because the SCO compiler will not inline a function with EH tables. Such functions tend to be complex, and by ignoring the request to inline them, the compiler is often doing the programmer a favor. The Haney numbers are broken down by subtest to show that in two of the three cases, the EH-enabled code runs *faster* than the EH-suppressed code, due to this effect.

The total size overhead numbers vary more widely, upwards of 20% or more in the worst cases. The bulk of this is the increased data in the three special `.eh_*` sections. The amount of EH table entries depends on lot on the object usage pattern in an application; a lot of objects being created and destroyed in short intervals tends to create larger tables, as do many small functions containing destructible objects. In any case, the `.eh_*` sections are not referenced until and unless needed.

How do these figures compare with other compilers?

§ ftp://ftp.cs.rpi.edu/pub/vandevod/Valarray

§§ http://webnz.com/robert/nzc_nm09.html

* ftp://ftp.cs.rpi.edu/pub/stl

3. Certain tests had to be dropped from the execution because they did not work with an up-to-date compiler, but the code sizes reflect all tests.

		EH suppressed			EH enabled			Overhead		
	SLOC	text	Σ size	time	text	Σ size	time	text	Σ size	time
Figure 4	50	344	351	0.77	344	351	0.77	0%	0%	0%
richards	.5K	3732	7384	6.79	4596	9397	7.30	23%	27%	8%
OOPACK	.7K	5652	126356	70.4	5652	126356	70.8	0%	0%	1%
valarray	1.1K	11332	16098	16.16	11844	17195	16.71	5%	7%	3%
deltablue	1.5K	8884	12666	1.11	9636	15467	1.15	8%	22%	4%
Haney	6K	40916	47167		40052	50572		-2%	7%	
	complex			17.0			16.95			- <1%
	real			18.62			18.1			-3%
	vector			18.72			19.94			7%
Newmat	8K	116980	145382	1.10	125508	180939	1.14	7%	24%	4%
HP STL	12K	253956	294906	0.04	249540	310119	0.04	-2%	5%	0%
ixx	14K	120164	166954	0.44	123844	188315	0.46	3%	13%	5%
lcom	19K	84644	143132	2.73	95972	171489	2.86	13%	20%	5%
eon	38K	552212	696714	61.72	536836	699625	61.83	-3%	1%	<1%

Table 1. EH overhead in sample C++ code

There has been considerable speculation but fewer published results in this area, and those there are [Horstmann 95] [Shimeall 95] [Meyers 96] [Plauger 97] have tended to use artificial or private benchmarks or to report different things. Suffice it to say that it appears that the above figures are at least comparable to those of other compilers. Certainly the static table-driven approach results in faster times than the design alternative of dynamic registration. For example, early in SCO's EH work a prototype implementation using dynamic registration showed a 18% execution time overhead for the Newmat benchmark, a figure typical for that approach [EDG 97].

5.2 Measurement of the effect of the EH optimization

Next we look at the question of what would be the effect if the EH optimization previously described were not in place.

Table 2 shows the difference in text size, total EH data size, and execution speed for the sources, compiled in the default way ("Default", same as in Table 1), and compiled using an undocumented environment variable to suppress this particular EH optimization ("No Opt"). Intuitively, the default way should produce better numbers. The "savings" numbers represent the gain of the optimization: the higher the number, the better.

This time, the Haney numbers are given whole, as there were no significant variations among the subtests. The current Silicon Graphics STL implementation† is added, despite containing try blocks, because STL is so important to modern C++.

What do these numbers show? The optimization produces some good gains on EH data size, especially in the modern, template-based classes such as valarray and STL. Like many specialized optimizations, its effect on text size and execution time can be strong in an isolated piece of code, but overall tends to be modest or imperceptible (but in optimization, every percent helps). The efficacy of the optimization depends a lot upon an application's source code organization, with it doing well when there are inlines (STL has a lot of template inlines) or functions that reference other functions within a translation unit. In any case, the optimization is very inexpensive to perform, so no harm is done if it does not come up with anything.

5.3 Measurement of the effect of empty exception specifications

Now we address the third question. So far we have measured the effectiveness of the optimization upon source that does not use exceptions or exception specifications, i.e. source for which the compiler had to deduce which functions could not throw an exception. But what is the effect of the optimization upon code that *does* use exception specifications heavily?

There are few if any such sources publicly available. So instead, the C++-to-C++ instrumenting feature of the Edison Design Group compiler was used to produce versions of the benchmarks, that added empty exception specifications to every non-C function lacking an exception specification. Admittedly, this puts them in far more places (including virtual functions) than would be likely in real use. But this at least provides an upper bound for what the effects of adding empty exception specifications might be. This was not done for the benchmarks that use templates, however, because as a rule adding exception specifications for unconstrained template code (such as STL) is ill-advised [Müller 96].

Table 3 shows the text size, total EH data size, and execution speed for the modified sources, compiled both in the default way and without the optimization (as in Table 2). The "improvement" numbers represent the gain of adding the empty exception specifications compared to the unmodified sources, for both with and without the optimization; the higher the number,

† http://www.sgi.com/Technology/STL/

	Default			No Opt			Savings		
	.text	Σ .eh	time	.text	Σ .eh	time	.text	Σ .eh	time
Figure 4	344	0	0.77	440	232	0.95	28%	∞%	23%
richards	4596	1080	7.30	4596	1080	7.31	0%	0%	<1%
OOPACK	5652	0	70.8	5652	0	70.4	0%	0%	-1%
valarray	11844	572	16.71	12196	1132	16.82	3%	98%	1%
deltablue	9636	2032	1.15	9636	2032	1.15	0%	0%	0%
Haney	40052	4252	54.99	40228	4292	55.0	<1%	1%	<1%
Newmat	125508	28648	1.14	125780	28852	1.13	<1%	1%	-1%
HP STL	249540	19596	0.04	251316	22476	0.04	1%	15%	0%
SGI STL	218884	29104	0.78	220308	31344	0.77	1%	8%	1%
ixx	123844	17664	0.46	123860	18328	0.45	<1%	4%	-2%
lcom	95972	16940	2.86	96036	17240	2.79	<1%	2%	-2%
eon	536836	17664	61.83	536148	18124	63.12	<1%	3%	2%

Table 2. Effect of EH optimization in sample C++ code

	Default			Improvement from normal			No Opt			Improvement from normal		
	.text	Σ .eh	time	.text	Σ .eh	time	.text	Σ .eh	time	.text	Σ .eh	time
Figure 4	360	0	0.83	-5%	0%	-8%	476	472	2.35	-8%	-∞%	-147%
richards	4580	1160	6.74	<1%	-7%	8%	5940	4040	14.95	-29%	-274%	-105%
OOPACK	5524	120	79.3	2%	-∞%	-8.9%	5568	1600	126.5	1%	-∞%	-80%
deltablue	9028	1640	1.08	6%	19%	6%	10612	6112	1.37	-10%	-201%	-19%
ixx	124660	22984	0.50	-1%	-30%	-9%	127396	57848	0.58	-3%	-216%	-29%
lcom	95044	17112	2.78	1%	-1%	3%	98388	40120	3.00	-2%	-133%	-7%

Table 3. Effect of adding `throw()`'s to all functions

the better. If the improvement is negative, adding the exception specifications degraded performance.

What do these numbers show?

- That when empty exception specifications are added and the optimization is in place, there can sometimes be significant savings over not using exception specifications at all, such as with richards and deltablue, where execution times become about the same as the EH-suppressed numbers of Table 1. In other cases there can be significant degradations[4].
- But when empty exception specifications are added and the optimization is *not* in place, the generated code becomes much worse than if the specifications were not there at all! Execution times get worse by up to a factor of two, and EH data sizes by up to a factor of three.

Thus when exception specifications are present, this optimization is of critical importance. While exception specifications were not explicitly designed with the goal of giving compilers the ability to better optimize [Koenig 90], that has been an expectation, at least among the C++ user community[5] and compiler vendors.[6] Without this compiler optimization, adding exception specifications to code does indeed *de-optimize* it, as Table 3 shows. This can be surprising to programmers [Meyers 96].[7] With it, exception specifications cause no extra overhead unless necessary, and the optimization that they do produce can be propagated further upwards.

Note however that exception specifications, including empty ones, should not be used casually in an effort to achieve a performance boost. [Reeves 96] presents a number of reasons why proper use of exception specifications can be problematic. Nevertheless, as programmers make use of exception specifications, and as some of the new C++ Standard library functions that have exception specifications are used, this optimization will become more and more important.

6. Areas for Future Work

This paper suggests several ways in which the elimination of EH tables and their associated indirect inefficiences could be

4. However the slightly worsened numbers for the Figure 4 default case are due to code generator weirdness and not to EH tables getting introduced.
5. For example, see the discussions of this topic in the Usenet newsgroup comp.lang.c++.moderated during November 1996 and February-March 1997.
6. During the C++ language standardization process Sun Microsystems and others argued successfully for strengthening the semantics of exception specifications such that these optimizations *could* be made.
7. Indeed, during ANSI public review comments on the draft standard, a request was received for a new language feature to overcome the fact that a commercial compiler was experiencing this de-optimization for such code.

improved: analysis of functions in two passes; dynamic flow analysis of virtual and indirect function calls; analysis of non-empty exception specifications (perhaps more useful in Java); identifying POSIX and other standards-based C calls known (under a compiler option) not to thrown an exception; elimination of try blocks from instantiated functions that cannot throw an exception; and inter-compilation-unit analysis.

7. Conclusions

Unlike many C++ compilers, the SCO compiler does not offer a user-level compilation option to suppress exception handling[8] (or any other language feature). The philosophy is that it is a compiler vendor's job to provide an implementation that makes every language feature acceptable, in terms of performance or any other criterion. Thus it is especially important that the overhead resulting from exception handling not be objectionable.

The measurements given here show that the general overhead, while quite variable from one application to the next, can indeed be kept down to acceptable levels.

Furthermore a specific, inexpensive optimization to detect functions that do not need to have EH information generated for them, can provide a modest benefit on some general code and a critical benefit on code that uses empty exception specifications.

Acknowledgments

Portions of the SCO exception handling implementation are derived from the Edison Design Group C++ compiler product, which uses a dynamic registration approach but allows a number of its EH data structures to be reused in a table-driven approach.

Joel Silverstein, Susan Carvalho, Paul Putter, and Robert Geva contributed to the design and implementation of exception handling support in the linker and compiler back end.

Daveed Vandevoorde, Glen McCluskey, Elaine Siegel, Joel Silverstein, Dave Prosser, and Steve Adamczyk contributed useful comments on all or parts of this paper.

References

[ABI 93] *System V Application Binary Interface, Intel 386™ Architecture Processor Supplement* (Third Ed.), UNIX Press, 1993.

[C++ 97] *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++*, X3J16/97-0108 WG21/N1146, 25 Nov. 1997.

[Cameron 92] Cameron, D., P. Faust, D. Lenkov, and M. Mehta, "A Portable Implementation of C++ Exception Handling", Proc. USENIX C++ Conference, August 1992.

[Chase 94] Chase, D., "Implementation of exception handling-I", Journal of C Language Translation, Vol. 5, No. 4, June 1994.

[Christensen 95] Christensen, M.M., "Methods for Handling Exceptions in Object-oriented Programming Languages", M.Sc. Thesis, Department of Mathematics and Computer Science, Odense University, January 1995.

[Dean 96] Dean, J., G. DeFouw, D. Grove, V. Litvinov, and C. Chambers, "Vortex: An Optimizing Compiler for Object-Oriented Languages", Proc. OOPSLA '96, ACM SIGPLAN Notices, Vol. 31, No. 10, October 1996.

[Driesen 96] Driesen, K., and U. Hölzle, "The Direct Cost of Virtual Function Calls in C++", Proc. OOPSLA '96, ACM SIGPLAN Notices, Vol. 31, No. 10, October 1996.

[EDG 97] *C++ Front End Internal Documentation*, Edison Design Group, Inc., February 1997.

[Horstmann 95] Horstmann, C.S., "C++ compiler shootout", C++ Report, Vol. 7, No. 6, July-August 1995.

[Koenig 90] Koenig, A., and B. Stroustrup, "Exception Handling for C++", Journal of Object Oriented Programming, Vol. 3, No. 2, July/Aug. 1990.

[Kristol 86] Kristol, D.M., "Four Generations of Portable C Compiler", Proc. USENIX Summer Conf., 1986.

[Lajoie 94] Lajoie, J., "Exception Handling — Supporting the runtime mechanism", C++ Report, Vol. 6, No. 3, March-April 1994.

[Meyers 96] Meyers, S., *More Effective C++*. Addison-Wesley, 1996.

[Müller 96] Müller, H.M., "Ten Rules for Handling Exception Handling Successfully", C++ Report, Vol. 8, No. 1, January 1996.

[Plauger 97] Plauger, P.J., "Embedded C++: An Overview", Embedded Systems Programming, Vol. 10, No. 12, December 1997.

[Reeves 96] Reeves, J.W., "Ten Guidelines for Exception Specifications", C++ Report, Vol. 8, No. 7, July 1996.

[Shimeall 95] Shimeall, S.C., "An Exception Hierarchy for Embedded Applications", Embedded Systems Programming, Vol. 8, No. 11, Nov. 1995.

[Stroustrup 93] Stroustrup, B., "A History of C++", SIGPLAN Second History of Programming Languages Conference (HOPL-II), ACM SIGPLAN Notices, Vol. 28, No. 3, March 1993.

[Stroustrup 94] Stroustrup, B., *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Stroustrup 97] Stroustrup, B., *The C++ Programming Language (Third Ed.)*. Addison-Wesley, 1997.

8. Except in the narrow case where C++ is used to write kernel-level device drivers, in which context use of EH is not allowed.