C++ Benchmarks in SPEC CPU2006

Michael Wong
IBM
michaelw (at) ca.ibm.com

Abstract

In SPEC CPU2006, there are three C++ integer benchmarks and four floating-point C++ benchmarks. This paper describes the work of incorporating C++ benchmarks into SPEC CPU2006. It describes the base language standard supported and the basis for run rules adopted to maintain an even playing field for different compilers. It also describes issues that complicate porting C++ benchmarks. It describes some of the C++ Standard compliance issues that were technically interesting during the benchmark development phase, using as examples the behavior of const-correctness, nested class access of private member of enclosing class, and unneeded template instantiations.

C++ Benchmark Selection Challenges

SPEC CPU2006 has seven C++ benchmarks compared to SPEC CPU2000's one. The committee viewed it as essential that these benchmarks be ported in a way that was fair to all compilers and would allow the code to conform to the official C++ Standard. This paper describes some of the characteristics of these C++ benchmarks, and gives an account of the C++ Standard-specific issues that were encountered during the porting of some of these benchmarks. It also describes specific conditions that need to be satisfied to port other C++ benchmarks in the future because of the intrinsic differences that C++ code has versus Fortran and C code. Much of these differences mean that porting adds an extra layer of complexity to what is already a difficult process.

Benchmarks are useful to people who buy computers. They want them to tell about performance on a set of target applications. They are useful to people who sell computers. They tell enough about the performance to get the buyer's attention. They are also useful when designing computers and compilers. They represent the important details of applications so that they can be used as concise distillations in the design process. Thus the author was interested in supporting the addition of not just more C++ benchmarks, but C++ benchmarks of quality and breadth, to represent recent changes in the usage of C++, and to be relevant for the future.

SPEC prefers to develop CPU benchmarks from actual end-user applications, instead of using synthetic benchmarks. Multiple vendors use the SPEC CPU suite and support it. It must be highly portable and as such the specific language standard seems to be the best way to enforce such portability.

A problem is that C++ as a Standard has many dark corners that make it difficult to enforce commonality across multiple compilers and platforms. The benchmark candidate source may contain errors. Furthermore, each compiler may have its own unique peculiarities based on its history, which complicates the task of determining correct, acceptable behavior. The easiest, most portable benchmarks may end up being the lowest common denominator of C++ applications. They would be too much like C, rather than testing features specific to C++. This would not benefit the C++ community as the benchmark candidates would use less interesting, less advanced, non-controver-

Table 1: C++ Benchmark Characteristics

Benchmark	Suite	Files	Bytes	Lines	Symbols	Comments
252.eon	Int 2000	326	1,222,099	41,188	1,527	The only C++ benchmark in SPEC CPU2000
471.omnetpp	Int 2006	155	1,352,414	47,910	1,428	It will encourage the OS to speed up malloc and may encourage optimizers to perform malloc optimization.
473.astar	Int 2006	20	108,557	5,849	176	It makes very little use of C++ features.
483.xalancbmk	Int 2006	1773	18,616,140	553,643	10,426	A large e-business app that also uses STL with very large data input set. It will push most stack memory to the limit while encouraging malloc improvement.
444.namd	FP2006	33	145,474	5,322	410	A good HPC benchmark but somewhat simple
447.dealII	FP2006	452	7,160,511	198,649	4,801	Uses Boost libraries and complex template techniques. Best representative of future C++ directions.
450.soplex	FP2006	124	1,157,229	41,435	195	Not very high on usage of C++ features
453.povray	FP2006	210	3,738,706	155,170	6,761	It is representative of C++ the way it is used currently. Has a potential single hot spot in the noise function

sial areas of the language, as well as rarely stressing the compilers involved.

SPEC CPU members were unwilling to settle for the lowest common denominator of C++ applications. The world of C++ is evolving rapidly beyond object-oriented programming of the '90s. Recent developments in generic programming and meta-template programming have shown C++ to be more capable than suspected. New programming styles such as expression templates allow C++ to expand to scientific computations traditionally reserved for Fortran. The Boost libraries [1] give C++ an experimental leading-edge arena to test new libraries developed by expert C++ programmers before their incorporation into the Standard.

C++ characteristics make its code look different from C code when viewed via intermediate representation to an optimizer. The code pattern that is generated through deep virtual functions, templatized inheritance, and meta-programming is different from that generated through C code. Even though we are reasonably assured of good code generation for basic cases such as simple virtual function calls, anything more complex is challenging to optimizers. A new set of strenuous C++ benchmarks supports the progression of this science for C++ applications, thus benefiting all C++ users.

SPEC CPU2000 has only one integer C++ benchmark, known as 252.eon. All others were C or Fortran benchmarks. This has now been improved in SPEC CPU2006, where there are three C++ integer benchmarks and four floating-point C++ benchmarks. Table 1 gives their names and a few comments about their nature. A more thorough description can be found at [2]. The first row in the table is the only C++ benchmark in SPEC CPU2000. All the following values show the increased size of the various benchmarks of SPEC CPU2006.

C++ Standard Updates

For the C++ language, the official standard is now ISO/IEC 14882:2003 [3] which is effectively the 1998 C++ Standard (ISO/IEC 14882:1998(E)) with the defect fixes applied through Technical Corrigendum 1 (TC1). Although it is commonly known as the 2003 Standard, programmatically it is still the 1998 Standard. That is, a program can check the macro `__cplusplus` and see that it still retains the old value, which is `199711` (because it was approved in Nov 1997). The reason is that the TC1 additions are mostly relaxing changes from the 1998 Standard: programs that used to fail can pass if they are valid by the 2003 changes.

Vendor compliance is a matter of corporate policy and the market that they serve. Some have moved their code to 2003. Some have moved beyond it by addressing defects that are accepted in the C++0x draft while others have stayed much closer to the 1998 Standard. This forms one of the difficulties of porting C++ and the reason some latitude is needed because different compilers may position themselves on different parts of the continuum of conformance.

Currently, the C++ Standard Committee is working towards revising the Standard for 2009. This is tentatively named C++0x [4]. A number of Working Papers propose ideas such as Concurrency [5], Concepts [6], and Garbage Collection [7] to be added into C++0x. These features could significantly change the way C++ is programmed in the future. The C++ Standard committee is also considering other key improvements to the language to keep it competitive with other languages that have evolved since its last Standardization in 1998. This work will likely impact the way C++ is programmed for the next ten years and the relevance of C++ in SPEC CPU benchmarks.

SPEC references language standards in the CPU2006 Run Rules [8] in Section 2.2.1, the "safety" rule, which was updated for CPU2006 to reference C99, Fortran 95, and C++98. Because of differences in compiler practices, and because the C++ Standard is programmatically still the 1998 Standard, SPEC chose to reference the 1998 Standard for C++. Under the safety rule, if a compiler can be shown to be conformant to the referenced standard, then that is considered an adequate defense of safety. Note that SPEC will also accept later standards than the one referenced, so if a compiler behavior conforms to the ISO-published C++2003, that is also an adequate defense of safety.

Leveling the Playing field with RTTI

Another SPEC CPU2006 rule change relating to C++ should be discussed. Section 2.2.1 says:

> *Note that for C++ applications, the standard calls for support of both run-time type information (RTTI) and exception handling. The compiler, as used in base, must enable these.*
>
> *For example, a compiler enables exception handling by default; it can be turned off with* `--noexcept`*. The switch* `--noexcept` *is not allowed in base.*
>
> *For example, a compiler defaults to no run time type information, but allows it to be turned on via* `--rtti`*. The switch* `--rtti` *must be used in base.*

This rule was adopted to create an even playing field for different compilers with different defaults on accepting Runtime Type Identification (RTTI) and Exception Handling. One of C++'s guiding design principles, as described in Bjarne Stroustrup's *Design and Evolution of C++*, [9] is the zero-overhead principle:

> *What you don't use, you don't pay for (zero-overhead rule).*

Stroustrup wants to avoid:

> *the overhead of supporting supposedly advanced features [which] is distributed over all the features in the language. For example, all objects are large to hold information needed for various kinds of housekeeping.*

A C++ feature should introduce no overhead into programs that do not use it.

> *If in doubt, provide means for manual control.*

Although the zero-overhead rule is not codified in the C++ Standard, its spirit has been retained by many compilers. It said that most compilers should offer a switch to turn on/off these features. In practice, the industry support varies: some compilers default to on, and some compilers default to off. But in most cases, some kind of control over exception handling and RTTI is offered because they are deemed to be the easiest to remove, and have the most beneficial result from manual control. Some other features cited by Stroustrup – e.g. virtual functions, multiple inheritance, and templates – rarely have manual control because they are either too tightly integrated into the compiler and thus hard to offer manual control, or they have little benefit from removal anyway, since there is no detrimental effect if the user does not invoke them directly. For example, unless the user declares a virtual function, or template instantiation mechanism, the overhead machinery, which in these cases is the virtual function table, and the code bloat from multiple instantiations, will not be there.

But exception handling and RTTI do incur overhead even if not used. For exception handling, extra state information or range values must be maintained. For RTTI, additional extended type information has to be added to the virtual function table or class bodies. These waste either space or code and may do so even if no code relating to the feature exists. So vendors have been known to offer them as default to be off or on, as their implementation design requires. Almost every compiler offers this control. Part of the problem is that this default state is not uniform.

For example, the IBM® XL C/C++ Enterprise Edition V8.0 for AIX® compiler [10] by default always has exception information on, because an exception mechanism usually exists in most standard library headers and thus cannot run without it. But RTTI is turned off by default so that unless the user uses one or more of `dynamic_cast`, `static_cast`, `const_cast`, `reinterpret_cast`, or `typeid`, the extra information is not built.

In practice, the waste in space may be small depending on the design, but it is possible for this to affect run results. The SPEC CPU subcommittee has consciously leveled the playing field by mandating these states to be on in base as a rule. This rule avoids the unfair case where some compilers simply have no way of turning RTTI or Exception Handling off. Note that the C++ Standard mandates the availability of these features so a compliant compiler must make them available.

Porting a C++ Benchmark

Surprisingly, most programs are not good benchmark candidates regardless of the language. Some examples of problems are workloads that cannot be consistently and easily verified, or systems intrinsically tied to some non-standard display code, or benchmarks that have one hot loop that can be too easily targeted for a narrow optimization. These common characteristics rarely receive much favorable support from the majority of SPEC members.

C++ adds additional constraints. The complexity of the C++ language makes most compilers' level of correctness and conformance non-uniform. One compiler may conform to the 1998 Standard, another may conform to the 2003 level, while a third may move support beyond that. All compilers also tend to add extensions to support their set of customers. The most problematic of these are peculiarities in the compiler that may be considered as bugs, or at least as historical artifacts, in addition to such bugs or artifacts in the benchmark source code. This means that porting will often have to battle potentially incompatible compiler peculiarities, and the more complex the code, the more likely that these will become the biggest problem. Consequently, this was the largest class of problems in porting C++ benchmarks.

C++ also has a predominant theme of using libraries; but SPEC prefers to use a standalone executable. Apache Xalan-C++ is one such library that supports text to XML to HTML transformations. It contains an XML parser based on Version 2.5 of Xerces-C++. The benchmark version, 483.xalancbmk, uses the XalanTransform program, which applies an XSLT stylesheet file to an XML document file and writes the transformation output to an output file. This was based on Version 1.8 of Apache Xalan-C++.

The original code was already quite portable as the authors maintained the port using macros and platform-specific files as needed. The major issue was the benchmark data. Most of the work of developing the benchmark was in finding appropriate data of acceptable content and length to create a desired running time.

483.xalancbmk was the result of packaging Xalan-C++ into an executable program and using as input an appropriately large XML file and the correct stylesheet. The issue with benchmark data is because parsing is such an efficient process that most XML transformation occurs rapidly even though verification was turned on, which effectively reprocesses the input. The quality and runtime of benchmark data is one of the major problems with any benchmark. The runtime must be sufficiently large today to take account of future machine speed improvement. On the processors used during suite development, the runtime turns out to be about 20-30 minutes for reference data, 25% of the reference time for train data, and less than 2 minutes for test data. A large XML sample was actually hard to find. Some of the largest publicly available XML files were the Shakespeare public domain files and even these ran within three minutes, which was still insufficient. Instead, an XML generator was used to generate a 50 Megabyte file, which ran in about 30 minutes on a (year 2004 vintage) IBM pSeries® POWER5™ system.

The train data is used for training for Profile-Directed Feedback runs. This enables the compiler to learn typical branch choices and optimize for the most likely path. It usually should not be a subset of the reference dataset. So 483.xalancbmk uses *The Tragedy of Antony and Cleopatra*, which was made available by Moby Lexical Tools [11].

The library deal.II [12] is a C++ library targeted at adaptive finite elements and error estimation. It makes interesting use of the latest C++ programming techniques. This includes template programming and the Boost C++ library [1]. It uses elements of the Boost library on preprocessor, `shared_ptr`, `tuple`, `utility`, `static_assert`, and `type_traits`. Boost libraries are high-quality peer reviewed libraries mostly written by members who participate on the C++ Standard committee. They are designed to be platform-independent, but tend to rely on conformance to the darker corners of the C++ Standard. The code is actually quite standard compliant, but not every compiler is fully aware of all the subtle corners of the Standard. This makes it also one of the most difficult benchmarks to compile because each compiler has different amounts of support of the template section of the C++ Standard. The acceptance of this benchmark should promote better long-term optimization of these unique C++ features. These template techniques are beginning to find their way into mainstream programming methodologies as well as numerical methods.

The remainder of this paper describes some of the C++ Standard-compliant issues that were technically interesting during the benchmark development phase and illustrate these code examples and their resolution. These issues include (1) the behavior of const-correctness, (2) nested class access of private member of enclosing class, and (3) unneeded template instantiations.

1. Const-correctness of library string returns:

This case is actually an error in the source code that can be compounded by errors in a C++ vendor's implementation of a C library function. This leads to invalid code being accepted by an invalid implementation, complicating the discovery of the problem. The original source is incorrect and needs to be changed. Compilers that compile the original code without issuing a fatal error will most likely also need to change their header file.

The problem: Code such as Figure 1 Part A, if compiled with a C++ 98 Standard-compliant header file, causes the expansion shown in Figure 1 Part B, because Standard section 21.4 paragraph 10 says [white space adjusted for emphasis]:

The function signature strstr(const char, const char*) is replaced by the two declarations:*

```
const char* strstr(const char* s1,
                   const char* s2);
      char* strstr(      char* s1,
                   const char* s2);
```

both of which have the same behavior as the original declaration.

This leads to the error shown in Figure 1 part C.

The rationale: If a compiler compiles this without error – and a few did – it is exposing a security hole. This causes the code to return a non-const pointer into the first argument (which came in as a const qualified type in the first argument), and now we can possibly maliciously change it through `f` because `f` is non-const qualified.

C++ 98 compliant compilers issue a severe error indicating that assignment into `f` causes a const to be assigned into a non-const, which is an error because, by allowing a non-const pointer to reference the data of a const pointer, we are violating an implied constraint of the const pointer. This would create a security hole because we are granting greater permission than the data declaration specifies. The Standard permits an increase in constness in conversions but not a decrease.

The C++ Standard in section 21.4 paragraph 10 indicates that the header file should have declaration for `strstr` functions that preserve const safety. When the first prototype is used, it preserves const safety (return is const and first argument is const) and when the second prototype is used, the input is non-const anyway so the return can be non-const.

The code fix: Code of this form can be trivially fixed by adding a const qualifier to the returned type, as shown in Figure 1 Part D. This fix will not break the conforming compiler

Figure 1: Const Correctness

A. The original code:

```
1    #include <string.h>
2    int main(){
3       const char * a="fefeeff";
4       const char * b="e";
5       char *f=strstr(a,b);
6    }
```

B. Expansion from C++ header file:

```
extern const char *strstr(const char *, const char *);
extern "C++" {
   inline char *strstr(char * __s1, const char *__s2){
      return (char *) strstr((const char *) __s1, __s2);
   }
}
```

C: Generated Error:

```
test.cc, line 5: Error: Cannot use const char* to initialize char*.
```

D: The fix:

```
int main(){
   const char * a="fefeeff";
   const char * b="e";
   #if defined(SPEC_CPU)
      const char *f=strstr(a,b);
   #else
            char *f=strstr(a,b);
   #endif
```

or the header file of compilers that don't conform to the Standard. They will continue to compile it as before.

Const-correctness is an often-neglected aspect of older code and is recently championed as the way towards greater security. The biggest problem is that it is viral, meaning that once started, it must be propagated down the call path, which often requires a large amount of code change. This cannot be avoided if compliance is required.

2. Nested class can access private member of enclosing class

Nested class access is disallowed in the C++ 1998 and 2003 Standard but is in the draft for the future C++0x [4] Standard through the Defect 45's proposed resolution [12] accepted in April 2001. Some compilers have moved aggressively ahead to accept some of these defects' proposed resolutions (likely due to customer demands). So even though the SPEC Run Rules only reference C++ 1998, it was decided that some latitude should be granted to accept this code modification using a macro to control compilation. The original C++ 1998 Standard says in Section 11.8 paragraph 1 (white space, comments adjusted in the excerpt):

The members of a nested class have no special access to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules (clause 11) shall be obeyed. The members of an enclosing class have no special access to members of a nested class; the usual access rules (clause 11) shall be obeyed. [Example:

```
class E {
  int x;
  class B { };
  class I {
    B b; // error: E::B is private
    int y;
    void f(E* p, int i)
    {
     p->x = i; // error: E::x is private
    }
  };
  int g(I* p)
  {
    return p->y; // error: I::y is pri-
vate
  }
};
```

This decision was reversed in the proposed resolution for Defect 45 and is part of the draft for C++0x:

A member of a class can also access all names as the class of which it is a member. A local class of a member function may access the same names that the member function itself may access. ... A nested class is a member and as such has the same access rights as any other member.

The test case: The test case is as follows:

```
class cQueue
{
  private:
    struct QElem
    {
      QElem *prev, *next;
    };
  public:
    class Iterator
    {
      private:
      QElem *p; // Error (?)
    };
};
```

Compilers that conform to the original 1998 Standard issue an error message indicating that p cannot access a private member. Compilers that have applied the proposed resolution for Defect 45 will compile this.

The code fix: This kind of code is actually common. It shows up in 453.povray and 471.omnetpp, and much newer code will tend to have this and expect compilers to allow it. The rationale indicates that it is intuitive. The agreed SPEC code fix is to add a macro to increase the class access of the private member to public for compilers that conform to the original Standard (i.e. which have not implemented the proposed resolution for Defect 45):

```
class cQueue
{
#if defined (SPEC_CPU_NO_NESTED_CLASS_AC-
CESS)
  public:
#else
  private:
#endif
    struct QElem
    {
        QElem *prev, *next;
    };
```

Admittedly, this is kind of a sledgehammer fix but there does not appear to be any other way around it. The disadvantage of this fix is that it exposes private data members to the world by making them public. So in the real world, one should use this fix only for the references that require it rather than make everything public. Since we are dealing with a standalone benchmark, it is unlikely to pose a security risk

3. Some compilers instantiate template member functions too eagerly leading to code bloat.

During the porting of deal.II [13] to create the benchmark 447.dealII, it was discovered that some compilers instantiate certain virtual functions too eagerly, but that an allowed ambiguity in the C++ Standard permits this behavior. Templates in C++ are classes or functions that do not use actual types or values, but generic placeholders. Templates can later be "instantiated" by replacing the placeholder by a concrete type or value. In addition, programmers can provide "explicit specializations" of templates for certain cases if the arguments with which a template is instantiated match certain conditions.

The case in question with 447.dealII is how eagerly the compiler should instantiate a template when it sees the template used with a concrete argument. C++ implicit instantiation is based on lazy instantiation, i.e. the language mandates that compilers only instantiate what it has to and leave as much as possible uninstantiated until there is no choice. The rules that codify this behavior are purposefully ambiguous in some respects, however. For demonstration of one of the corner cases consider this code:

```
1 template <typename T> struct A {
2     typedef int type;
3     virtual void foo ();
4 };
5
6 template <typename T> void A<T>::foo () {
7   T().compilation_yields_an_error();
8 }
9
10 template <typename T> struct Unrelated {
11    void foo (const typename A<T>::type)
const;
12 };
13
14 template <> void Unrelated<int>::foo
15             (const A<int>::type) const;
```

Line 14 contains the declaration of an explicit specialization of function template "Unrelated" with template parameter `int` using call parameters "`const A<int>::type`". In other words, it tells the compiler that a definition of this function for the template argument `int` can be found somewhere else. In order to see what the type "`A<int>::type`" refers to, the compiler has to implicitly instantiate "`A<int>`".

When a class template is implicitly instantiated, the declarations of its members are instantiated as well, although the corresponding definitions are not (i.e. the function bodies are not compiled). However, there are a few exceptions to this. First, if the class template contains an anonymous union, the members of that union's definition are also instantiated. A second exception concerns default function arguments. The third exception (and in this context the only one that is relevant) deals with virtual member functions: their definitions may or may not be instantiated as a result of instantiating a class template. Some implementations will, in fact, instantiate the definition because they instantiate the virtual function tables that enable the virtual call mechanism. Virtual function tables are lists of pointers to member functions. If they are instantiated and written into the object file, it is also necessary to instantiate the member function definitions, i.e. to compile the function bodies with template arguments substituted, to avoid undefined references in linkable entities. The instantiation of virtual function tables and thereby virtual function templates is specifically allowed by Paragraph 9 of Clause 14.7.1 of the C++ Standard:

An implementation shall not implicitly instantiate a function template, a member template, a non-virtual member function, a member class or a static data member of a class template that does not require instantiation. It is unspecified whether or not an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated. [...]

The reason this ambiguity exists may be because by the time this paragraph was put to words, there remained some implementations that could and would instantiate the virtual member functions. So in order to not break with previous legacy implementation, and since one of the considerations that Standards have to take into account is existing practice, it was felt that there was no need to specifically further restrict this paragraph.

On the other hand, in the program above, the instantiation of the virtual function table entails the instantiation of "`A<int>::foo`" which fails because line 7 does not make sense for "`T=int`". In the original program, this was no problem since a specialization for "`A<int>::foo`" was provided later on, but not before the time it was needed in line 14 by some compilers.

The deal.II library is a clever template program that supports finite element computations at 1d, 2d, and 3d. The elegance of generic programming allows one to write functions independently of the space dimension in order to test with cheap simulations in 1d or 2d, and later run realistic 3d simulations with the same code. The generic part of the code looks like this:

```
template <int dim> class Triangulation {
  typedef int local_type;
  void foo (local_type);
  virtual void bar ();
};
```

In some cases, however, 1d, 2d, and 3d algorithms and data structures may differ, and are implemented as explicit specializations. The SPEC benchmark 447.dealII uses 3d, but declarations of explicit specializations for the 1d and 2d case remain from the original deal.II library, as in

```
template <> void Triangulation<1>::foo
(Triangulation<1>::local_type);
```

The problem is that these explicit specializations trigger the case discussed above, i.e. some compilers may want to instantiate "Triangulation<1>::bar" or yield linker errors instead. The necessary function bodies exist in the deal.II library, but are guarded by preprocessor #if directives, based on the dimension case:

```
#if deal_II_dimension == 1
  template <>
  void Triangulation<1>::bar() {
                  do_something(); }
#else
  template <int dim>
  void Triangulation<dim>::bar () {
                  do_something_else(); }
#endif
```

This means that in a 2d or 3d build of 447.dealII, the definition for the 1d case is not available because the authors assumed it was not needed. Compilers that instantiate the virtual function table for the 1d "`Triangulation`" class need it, however.

There are three solutions to this problem. (1) Alter all of 447.dealII to remove the declarations of explicit specializations. The disadvantage of this approach would be a tremendous amount of editing. (2) Include the definitions of the 1d and 2d specializations as well, i.e. remove the #if guards. This, again, would require much editing and furthermore would force compilers that do not instantiate virtual function tables to go through thousands of additional lines of code, leading to code bloat. (3) Build the program in three passes, the first pass setting the variable `deal_II_dimension` to 1, the second pass to 2, then 3. In this way the unresolved references can be satisfied by the Explicit Specialization machinery, although no 1d or 2d function definition will ever be called. Solution 3 was chosen because it was fast and it was more portable to all compilers that need this approach. The default is to build once, but the triple compilation can be enabled by adding the following variable to the 447.dealII portion of a SPEC config file:

```
    explicit_dimensions = 1
```

The result is an executable that contains not only the 3d functions, but also function instantiations for 1d and 2d, even though the latter are never called. Consequently, this method does not produce a different runtime behavior; hence the difference in build is performance neutral.

There are other portability challenges for 447.dealII. The benchmark uses cutting-edge, standard-compliant C++ code, which means not all compilers can compile it. The state of C++ compiler support for the template section of the C++ Standard varies. This is partly the result of the changes injected and the complexity of the changes such that each compiler may interpret them differently. The SPEC config file can include flags that affect various capabilities and known workarounds if a compiler is known to be unable to support a particular construct. This is described in the text file `447.dealII/Docs/447.dealII_Config.txt.`

Conclusion

As shown in this paper, the porting of C++ benchmarks is complicated by the fact that compiler implementations can be incorrect, which leads to difficult-to-discover violations (case 1, above); there are ongoing changes in the C++ Standard that allow code to be acceptable in multiple forms (case 2); or there could even be ambiguity in the Standard that leads to conformant but multiple compilation possibilities (case 3). The SPEC CPU benchmarks must work within these known boundaries. SPEC CPU2006 has done an exceptional job of following these somewhat difficult parameters and producing a benchmark suite that is still performance neutral.

Most important is that SPEC CPU2006 has not taken the lowest common denominator of C++ code, which would have made it easier to get the widest portability but rendered SPEC CPU2006 irrelevant as a measurement tool for decision making. Instead, it has taken on the challenge of including a spectrum of C++ code from basic inheritance to generic programming to template programming in order to ensure coverage of most of the techniques that will be used in real code for the near future. This allows C++ to continue to enjoy that rare community of researchers and in-the-trenches programmers who actively collaborate to update the language. Not many other languages have such an active evolutionary focus.

Acknowledgements

Much of this paper would not have been possible without the encouragement of Alan Mackay. I also thank the following individuals who made valuable contributions during the SPEC benchathon and recommendations during the preparation of this paper: Christopher Cambly, Robert Klarer, Sasha Kasapinovic, Yan Liu, Wolfgang Bangerth, Jim McInnes, June Ng, Jeff Hamilton, Neil Graham, Andrew Godbout, Roland Koo, Anne James, Alex Ross, Steve Clamage, Darryl Gove, and John Henning.

The views presented in this paper are those of the author and not of IBM Corporation or any of its affiliates.

References

[1] http://www.boost.org
[2] J. Henning (ed.), "SPEC CPU2006 Benchmark Descriptions", Computer Architecture News, Volume 34, No. 4, September 2006. Also posted at www.spec.org/cpu2006
[3] The C++2003 Standard is ISO/IEC 14882:2003(E), available via http://webstore.ansi.org/ansidocstore/.
[4] C++0x: http://www.artima.com/cppsource/cpp0xP.html
[5] Concurrency: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1942.html
[6] Concepts: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2081.pdf
[7] Garbage Collection: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1943.pdf
[8] http://www.spec.org/cpu2006/Docs/runrules.html
[9] The Design and Evolution of C++, Bjarne Stroustrup, Addison Wesley 1994, Pg. 121
[10] http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp
[11] Moby: http://www.cis.strath.ac.uk/~dce/MIA/assess/data/data_set_01/shaksper.htm
[12] See the C++ Standard Core Language Active issues, http://anubis.dkuug.dk/jtc1/sc22/wg21/ docs/cwg_defects.html#45
[13] http://www.dealii.org/

Trademarks

Notices